ETR'05

ÉCOLE D'ÉTÉ TEMPS RÉEL 2005

4ème édition

ACTES

13 au 16 septembre 05

NANCY - Ecole Nationale Supérieure de Géologie (INPL)

http://etr05.loria.fr/

lorraine conseil régional

Meurthe & Moselle Conseil Général

Conception : Service communication LORIA-INRIA Lorraine

Loria

CNRS Centre National de la Recherche Scientifique

inpl nancy

INRIA

Université Henri Poincaré Nancy 1

Nancy 2 Université

ARP

QSL

Grand Nancy

Préface

C'est un grand plaisir pour nous de vous accueillir à Nancy du 13 au 16 septembre 2005 pour l'École d'été Temps Réel 2005 (ETR'05). Cette manifestation, soutenue par le GDR ARP (Architecture, Réseau et système, Parallélisme) du CNRS et co-organisée par l'INRIA Lorraine et le LORIA, est un événement important et bien établi au sein de la communauté temps réel francophone car c'est à Nancy la quatrième édition, après Poitiers en 1997 et 1999, et Toulouse en 2003.

L'objet de cette école est de rassembler étudiants, chercheurs et industriels pour faire le point sur les avancées des méthodes, techniques et outils dans le domaine des systèmes temps réel et de présenter leurs possibilités, limites et complémentarités.
Concrètement, les enjeux sont considérables car l'informatique temps réel, c'est à dire une informatique qui garantit à l'utilisateur un certain niveau de performances temporelles, comme un délai de réaction, est de plus en plus présente dans notre société, que ce soit pour offrir davantage de sécurité au conducteur d'un véhicule, piloter des centrales électriques ou des lignes de fabrication, ...

Le programme de ces 4 journées est construit autour d'exposés de synthèse donnés par des spécialistes du monde industriel et universitaire qui permettront aux participants de l'ETR, et notamment aux doctorants, de se forger une culture scientifique dans le domaine. Cette quatrième édition est centrée autour des grands thèmes d'importance dans la conception des systèmes temps réel :

- Langages et techniques de description d'architectures,
- Validation, test et preuve par des approches déterministes et stochastiques,
- Ordonnancement et systèmes d'exploitation temps réel,
- Répartition, réseaux temps réel et qualité de service.

Par ailleurs, deux ateliers montreront les possibilités d'outils logiciels pour la résolution de problèmes concrets. Notons également que les premières « Rencontres des Jeunes Chercheurs en Informatique Temps Réel » (RJCITR'05) se dérouleront au sein de l'ETR et offriront un espace d'expression aux doctorants pour faire connaître leurs travaux et établir un dialogue avec des chercheurs plus confirmés.

Nous voudrions enfin adresser nos remerciements les plus sincères aux partenaires de l'ETR qui ont apporté un soutien financier et logistique important sans lequel cette manifestation n'aurait pu se tenir. Merci donc aux Universités Nancéiennes (INPL, UHP Nancy), aux collectivités locales et régionales (Communauté Urbaine du Grand Nancy, Conseil Général de Meurthe et Moselle, Conseil Régional de Lorraine), au laboratoire LORIA, à l'INRIA et au CNRS.

Bienvenue à l'Ecole d'été Temps Réel 2005 !

Nicolas Navet
Pour les comités de pilotage scientifique et d'organisation d'ETR'05

Comité de pilotage scientifique

- Jean Philippe BABAU (CITI, INSA de Lyon)
- Mamoun FILALI (IRIT, Toulouse)
- Hacene FOUCHAL (GRIMAAG, Pointe-à-Pitre)
- Laurent GEORGE (ECE, Paris)
- Serge HADDAD (LAMSADE, Paris)
- Guy JUANOLE (LAAS, Toulouse)
- Zoubir MAMMERI (IRIT, Toulouse)
- Radu MATEESCU (INRIA Rhône-Alpes)
- Nicolas NAVET (LORIA, Nancy) - Responsable scientifique d'ETR 2005
- Françoise SIMONOT-LION (LORIA, Nancy)
- Yvon TRINQUET (IRCCyN, Nantes)
- François VERNADAT (LAAS, Toulouse)

Comité local d’organisation

- Laurence BENINI (projet TRIO, LORIA)
- Anne-Lise CHARBONNIER (service Colloques/Missions, LORIA)
- Armelle DEMANGE (service Colloques/Missions, LORIA)
- Nicolas NAVET (projet TRIO, LORIA) - Responsable de l'organisation d'ETR 2005
- Ammar OULAMARA (projet MACSI, LORIA)
- Xavier REBEUF (projet TRIO, LORIA)
- Françoise SIMONOT-LION (projet TRIO, LORIA)
- Sabrina VERDENAL (service Colloques/Missions, LORIA)
- Olivier ZENDRA (projet TRIO, LORIA)

Partenaires de l’ETR

- Action Qualité et Sûreté du Logiciel (QSL) du LORIA
- Communauté Urbaine du Grand Nancy (CUGN)
- Conseil Général de Meurthe et Moselle
- Conseil Régional de Lorraine
- Groupement de recherche CNRS Architecture, Réseaux et systèmes, Parallélisme
- Institut National de Recherche en Informatique et Automatique (INRIA)
- Institut National Polytechnique de Lorraine (INPL)
- Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA)
- Université Henri Poincaré (UHP Nancy)

Table des matières

Thème 1 - Langages de Description d'Architectures, Techniques de Description Formelles et Semi-Formelles

UML2 et ses profils pour le temps réel **9**
Sébastien Gérard, Hubert Dubois, Huascar Espinoza (CEA-List, Gif-Sur-Yvette)

The Bossa framework for scheduler development **23**
Gilles Muller (EMN-INRIA, Nantes), Julia L. Lawall (University of Copenhagen, Denmark)

Les langages de description d'architecture (ADL) pour le temps réel **31**
Anne-Marie Déplanche, Sébastien Faucou (IRRCyN, Nantes)

Les ADL du point de vue de l'industrie **47**
Jean-François Tilman (AXLOG Ingénierie)

Thème 2 - Model-Checking Temporel, Vérification Probabiliste, Techniques de Tests

Elements of model checking **55**
Stephan Merz (INRIA-Loria, Nancy)

Automatic verification and conformance testing for validating safety properties of reactive systems **69**
Vlad Rusu, Hervé Marchand, Thierry Jéron (IRISA, Rennes)

An introduction to timed automata **79**
Patricia Bouyer (LSV-ENS, Cachan)

Vérification de programmes synchrones avec Lustre/Lesar **95**
Pascal Raymond (Verimag, Gières)

Model checking for probabilistic timed automata with digital clocks **105**
Marta Kwiatkowska, Gethin Norman, David Parker (University of Birmingham), Jeremy Sproston (Università di Torino)

Atelier logiciel : TIme petri Net Analyzer (TINA) ..... ♦
François VERNADAT (LAAS, Toulouse)

Atelier logiciel : Construction and Analysis of Distributed Processes (CADP) ..... ♦
Radu MATEESCU (INRIA, Lyon)

♦ : Supports distribués aux participants pendant l'ETR

Thème 3 - Ordonnancement Temps Réel, Exécutifs Temps Réel

Les systèmes d'exploitation temps réel **123**
Yvon Trinquet (IRRCyN, Nantes)

Conditions de faisabilité pour l'ordonnancement temps réel préemptif et non préemptif **135**
Laurent George (Ecole Centrale d'Electronique, Paris)

Analyse des temps de réponse et de la demande processeur en ordonnancement temps réel de tâches périodiques **151**
Pascal Richard, (LISI-ENSMA, Poitiers)

Méthodes de calcul de WCET (Worst-Case Execution Time) - état de l'art **165**
Isabelle Puaut (IRISA, Rennes)

Conception conjointe commande/ordonnancement et ordonnancement régulé **177**
Daniel Simon (INRIA Rhône-Alpes, St Ismier)

Le dimensionnement temps réel dans l'automobile - étude de cas ..... ♦
Jaime de Oliveira (VALEO VESL, Créteil)

Java temps réel – un état de l'art **195**
Marc Richard-Foy (AONIX, Paris)

Thème 4 - Répartition, Réseaux, Qualité de Service

Qualité de service dans les réseaux : problématique, solutions et challenges **207**
Zoubir Mammeri (IRIT – Toulouse)

Tutoriel Network Calculus et détermination de bornes de délai ..... ♦
Patrick Thiran (EPFL, Suisse)
Tutoriel disponible à l'adresse http://ica1www.epfl.ch/PS_files/NetCal.htm

Control task timing and quality of control **225**
Anton Cervin, Dan Henriksson, Bo Lincoln, Martin Andersson,
Karl-Erik Årzén (Lund University, Sweden)

Adaptation des applications distribuées à la qualité de service du réseau de communication **233**
Fabien Michaut, Francis Lepage (CRAN, Vandoeuvre)

Some real-time issues in wireless sensor networks **255**
David Simplot-Ryl (INRIA Futurs, Lille)

Les réseaux de terrain : des réseaux temps réel **261**
Jean-Pierre Thomesse (LORIA-INPL, Vandoeuvre)

Approches (m,k)-firm pour la gestion de la qualité de service temps réel **269**
YeQiong Song (LORIA-INPL, Vandoeuvre)

Qualité de service dans les réseaux sans fil **285**
Guy Juanole (LAAS, Toulouse), Thierry Val (EA ICARE, Blagnac)

Thème 1

Langages de Description d'Architectures, Techniques de Description Formelles et Semi-Formelles

UML2 et ses profils pour le temps-réel

Sébastien Gérard
CEA-List
CEA/LIST/DTSI/SOL/LLSP
91190 Gif Sur Yvette
Sebastien.Gerard@cea.fr

Hubert Dubois
CEA-List
CEA/LIST/DTSI/SOL/LLSP
91190 Gif Sur Yvette
Hubert.Dubois@cea.fr

Huascar Espinoza
CEA-List
CEA/LIST/DTSI/SOL/LLSP
91190 Gif Sur Yvette
Huascar.Espinoza@cea.fr

Résumé

UML compte maintenant parmi les langages de modélisation les plus répandus, enseignés et outillés pour le génie logiciel. Bien que langage de modélisation généraliste, UML a la capacité d'être adapté aux besoins d'un domaine particulier d'application au travers de la définition de stéréotypes, valeurs étiquetées et contraintes réunis dans un profil UML. C'est ainsi que UML s'est également répandu dans des domaines où initialement il n'aurait pas pu trouver sa place. On retrouve donc tout naturellement des propositions de méthode et d'outils basés sur UML et adressant le domaine particulier du développement d'applications temps-réel embarquées. Le but de ce papier est de montrer comment UML tient désormais une place au sein des langages de développement pour le temps-réel (TR).

UML2 étant maintenant disponible, l'objet de cet article est d'une part de décrire rapidement les capacités intrinsèques de UML2 à modéliser des applications TR, et d'autre part à décrire ses profils spécifiques pour le TR.

1. Introduction

UML compte maintenant parmi les langages de modélisation les plus répandus, enseignés et outillés pour le génie logiciel. Bien que langage de modélisation généraliste, UML a la capacité d'être adapté aux besoins d'un domaine particulier d'application au travers de la définition de stéréotypes, valeurs étiquetées et contraintes réunis dans un profil UML. C'est ainsi que UML s'est également répandu dans des domaines où initialement il n'aurait pas pu trouver sa place. On retrouve donc tout naturellement des propositions de méthode et d'outils basés sur UML et adressant le domaine particulier du développement d'applications temps-réel embarquées. Le but de ce papier est de montrer comment UML tient désormais une place au sein des langages de développement pour le temps-réel (TR).

UML2 étant maintenant disponible, l'objet de cet article est d'une part de décrire rapidement les capacités intrinsèques de UML2 à modéliser des applications TR, et d'autre part à décrire ses profils spécifiques pour le TR.

2. Le temps-réel en UML2

L'objet de cette section est de présenter brièvement les constructions de UML2 permettant de modéliser les aspects temps-réel d'une application.

2.1. La sémantique d'exécution de UML2

La sémantique d'exécution de UML2 repose sur deux principes fondamentaux :

1. Tout comportement d'un système modélisé est la résultante d'actions exécutées par des objets, dits objets actifs. Ces derniers incluent les comportements qui, en UML2, sont des objets et qui peuvent être actifs et ainsi coordonner d'autres comportements.
2. La sémantique comportementale de UML2 ne traite que de comportement de type dirigé par les évènements ou type discret.

Fort de ces deux principes, la définition des aspects sémantiques d'exécution de UML est organisée en une architecture à trois couches dont la dépendance se fait des couches les plus hautes vers les couches les plus basses.

La couche basse qui définie les bases sémantiques de UML est d'ordre structurelle. Ce point reflète bien le premier principe cité ci-avant, à savoir le fait que tout comportement dans un modèle UML est issu d'actions opérées par des entités structurelles. La seconde couche est, elle, d'ordre comportementale et elle définie trois sous domaines sémantiques de base : la communication entre les entités structurelles, le comportement interne des entités structurelles et les actions élémentaires qui constituent le plus fin niveau de description comportementale en UML. La troisième et dernière couche décrit le niveau le plus haut en terme de formalisme dédié à l'expression du comportement et elle consiste en des activités, des machines à état et des interactions.

La sémantique d'exécution de UML2 repose également sur le modèle de causalité simple suivant : les objets répondent à des messages générés par des objets exécutant des actions de communication. La réception d'un message par un objet déclenche l'exécution d'un comportement particulier associé au message reçu. La

façon selon laquelle un comportement est associé à un message dépend du formalisme de haut niveau utilisé pour décrire le comportement et n'est pas défini dans la spécification de UML. Il s'agit là d'un point de variation sémantique de la norme. Cependant, le modèle de causalité définit un lien entre comportement appelant et comportement appelé au travers de l'action spécifique *CallBehaviorAction* qui définie le mécanisme de passage des arguments de l'appelant vers les paramètres de l'appelé. Enfin, ce modèle de causalité, qui est purement procédural, peut être utilisé soit directement tel quel, soit en conjonction avec un modèle de type orienté-objet.

2.2. Modéliser le comportement

Du point de vue comportemental, UML propose principalement trois constructions : un langage d'action, des machines d'états et des diagrammes d'activité.

2.2.1. Les machines d'états

Les machines d'état UML2 offrent de nombreux concepts, tels que la notion d'état hiérarchique, composite, historique, organisé par nœuds de branches qui, combinés, couvrent la plupart des formalismes basés sur la notion d'état. On y retrouve à la fois des machines de Mealy et des machines de Moore.

Les machines d'états se présentent sous deux formes :

1. Les machines d'état comportementales (« Behavioral state machines ») – elles peuvent être utilisées pour décrire le comportement de n'importe quel élément de modèle.
2. Les machines d'état de protocole (« Protocol state machines ») – ce type d'automate permet de modéliser des protocoles d'utilisation, par exemple le cycle de vie d'un objet ou des contraintes sur l'ordre des appels aux opérations d'un objet. On peut également utiliser ce type de représentation du comportement pour décrire la dynamique des interfaces et des ports d'un composant.

La sémantique associée aux machines d'états est une variante objet de la machine d'état de Harel [9]. Sa description est de type opérationnel et elle repose sur une machine d'exécution virtuelle composée de (Figure 1) :

1. Une file d'attente d'évènements qui sert à stocker les instances d'évènements entrantes en attendant de les consommer.
2. Une politique de choix des évènements qui détermine l'ordre d'extraction des occurrences d'évènement contenues dans la file d'attente.
3. Un processeur à évènements qui exécute les traitements associés aux évènements en respectant la sémantique des machines d'états-transitions de UML et, en particulier, l'hypothèse d'exécution « Run-To-Completion ».

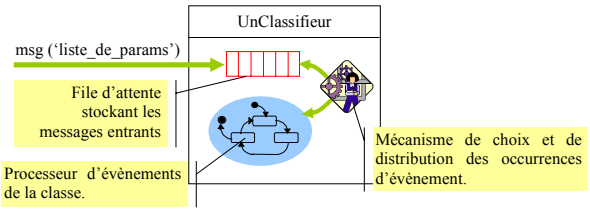


Figure 1. Sémantique opérationnelle du principe de la machine d'états de UML.

Les évènements sont dépilés un par un et consommés par une machine d'états-transitions. L'ordre dans lequel ils sont dépilés n'est pas précisé dans UML, cela constitue un point de variation sémantique. La sémantique d'exécution des évènements est basée sur l'hypothèse dite de « Run-To-Completion ». Cela signifie qu'un évènement ne peut être dépilé puis consommé que lorsque le traitement de l'évènement précédent est achevé. On parle de pas « RTC ».

2.2.2. Les diagrammes d'activité

Les diagrammes d'activité servent à modéliser des flux de contrôles et/ou de données. Il s'agit d'un graphe orienté de flux composé de nœud et d'arcs. La sémantique des diagrammes d'activité repose sur une circulation de jetons, proche de celle rencontrée dans les réseaux de Pétri. Un jeton modélise une donnée ou un objet. Sa circulation dans le réseau est conditionnée par les éléments de contrôle (arcs ou nœuds). Ces éléments permettent d'exprimer des notions de parallélisme et de synchronisation.

On distingue trois types de nœuds (Figure 2). Les nœuds d'action transforment les flux de donnée/contrôle d'entrée en flux de donnée/contrôle de sortie. Ces derniers sont alors les entrées d'autres actions. Les nœuds de contrôle définissent les règles de circulation des jetons à travers le graphe. Les nœuds objets servent à stocker temporairement des données ou des objets.

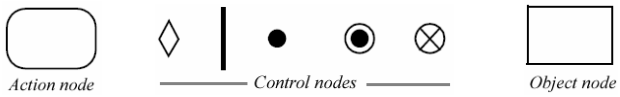


Figure 2. Type de noeuds des activités de UML2.

Pour connecter ces nœuds, il existe deux types d'arc : les arcs de flux de contrôle et les arcs de flux d'objet. Les premiers synchronisent le début d'une action (destination de l'arc) avec la fin d'une action (origine de l'arc). Les arcs de flux d'objet permettent de faire passer des valeurs entre deux nœuds.

2.2.3. Le langage d'action

UML2 défini le concept d'*Action* comme étant l'unité fondamentale de spécification comportementale permettant à des modèles UML d'être complètement exécutable. Le principe repose sur le fait que les actions

échangent des flux de contrôle et de donnée via des fiches d'entrée et de sortie.

On trouve quatre niveaux de concepts pour les actions : basiques, intermédiaires, structurés et complets.

Les actions du niveau basique définissent les fondements du langage d'action et fournissent principalement les mécanismes de base d'appel d'opération, d'envoi de signal et d'appel de comportement direct.

Les actions du niveau intermédiaire fournissent des constructions supplémentaires pour l'invocation de service et les premières actions de lecture et d'écriture.

Les actions du niveau complet défissent des actions supplémentaires de lecture et d'écriture et des actions diverses d'acceptation des évènements.

Le dernier niveau, le niveau structuré, spécifie les actions qui s'appliquent dans le contexte d'un nœud structuré ou d'une activité comme l'action de lever une exception.

Concernant le langage d'action, la spécification de UML ne définit qu'une syntaxe abstraite et une sémantique associée. [1] ne propose en effet pas de langage de surface permettant de manipuler directement les constructions d'action qu'il propose. Par conséquent, pour l'utiliser il faudra au préalable définir une syntaxe concrète du langage d'action.

Accord|$_{UML}$ [2, 3] est une méthodologie de modélisation d'application temps-réel. Elle repose sur des modèles UML non ambigus et complets au sens de l'exécutabilité. Pour ce faire, elle définit Accord|$_{AL}$ qui propose deux formalismes pour décrire les actions dans un modèle : un premier sous une forme textuelle du langage d'action de UML, un second sous une forme graphique basée sur les diagrammes d'activité et les actions de UML. Les deux formalismes sont équivalents et l'on peut basculer d'une vue à l'autre en fonction de ses besoins ou de ses habitudes de travail.

Dans [4], chaque action est définie de la manière suivante : sémantique (celle de la construction de base de UML), notation textuelle (au format EBNF[1]), équivalent graphique et à l'aide d'exemples illustrant l'utilisation de l'action. La syntaxe textuelle concrète des actions de boucle est ainsi définie par :

`<LoopAction> ::= <WhileLoop> | <ForLoop>` où les deux boucles classiques de type *`WhileLoop`* et *`ForLoop`* sont définies par :

```
<WhileLoop> ::= WHILE(<BooleanExpression>)
                    <GroupAction>
                ENDWHILE
<ForLoop> ::=  FOR(<ForCondition>)
                    <GroupAction>
               ENDFOR
```

L'exemple de la Figure 3 illustre une utilisation de Accord|$_{AL}$. Il s'agit de la modélisation d'un algorithme de régulation de la vitesse.

[1] EBNF = Extended Backus-Naur Form

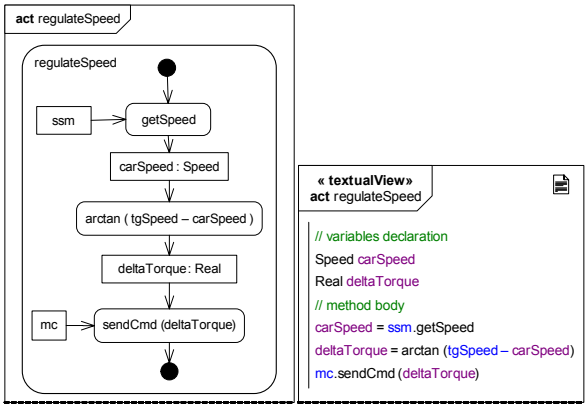


Figure 3. Vues graphique (partie gauche) et textuelle (partie droite) d'un modèle de traitement des données.

2.3. Modéliser les interactions

Une interaction est une construction de modèle qui permet de décrire une liste de messages échangés par un ensemble d'objets d'un système afin de réaliser une tâche donnée. Il s'agit donc de décrire un aspect dynamique d'un système de façon transversale.

2.3.1. Le diagramme de séquence

Le diagramme de séquence fait partie de la famille des diagrammes UML qui permettent de décrire des interactions. Les diagrammes de séquence associent à chacun des objets impliqués dans une interaction une ligne de vie verticale représentant le temps (le temps s'écoule de haut en bas). Sur cette ligne de vie, on peut placer des envois et des réceptions de messages échangés entre les objets. On forme ainsi une séquence ordonnée de messages. L'ordre est partiel par rapport à tout le système.

2.3.2. Le diagramme de communication

Tout comme le diagramme de séquence, le diagramme de communication permet de modéliser une interaction. Contrairement au diagramme de séquence qui permet de mettre l'accent sur l'aspect temporel (ordre des messages) d'une interaction, le diagramme de communication met l'accent sur l'aspect structurel d'une interaction.

Le diagramme de communication de UML2 correspond au diagramme de collaboration de UML1.x.

2.3.3. Le diagramme d'ensemble des interactions

Le diagramme d'ensemble des interactions (*Interaction Overview Diagram*) est une spécialisation du diagramme d'activité pour lequel chaque nœud correspond à un diagramme de séquence ou de communication. Le diagramme de communication permet de donner une vue d'ensemble d'un système en mettant l'accent sur le flux de données.

2.4. Le temps dans UML2

2.4.1. Un modèle basique du temps

Le modèle de temps de UML2 ne définit que des concepts de base pour supporter le temps :

1. *TimeExpression* définit la spécification d'une valeur qui représente une valeur de temps.
2. *Duration* définit la spécification d'une valeur décrivant la distance temporelle qui sépare deux expressions de temps marquant deux instants.

Deux actions spécifiques pour le temps sont également définies :

1. *TimeObservationAction* est une action qui lit la valeur du temps courante et l'écrit dans une valeur structurelle particulière.
2. *DurationObservationAction* est une action qui mesure une valeur de durée du temps et la stocke dans une valeur structurelle particulière.

Finalement, le paquetage de temps de UML2 définit le concept de contrainte d'intervalle : *IntervalConstraint*. Cela permet de décrire des contraintes portant sur des intervalles de temps.

2.4.2. Temps et machines d'états

Dans le contexte des machines à états, UML définit un événement spécifique appelé *TimeEvent*. Il sert à modéliser l'expiration d'une échéance qui peut être relative ou absolue :

1. Un évènement dénotant le passage d'une quantité de temps suite à l'entrée dans l'état contenant la transition est noté avec le mot-clé *after* suivi d'une expression de type *TimeExpression* qui donne la valeur temporelle de l'événement.
2. Un évènement dénotant l'occurrence d'une date absolue est noté via le mot-clé *when* suivi d'une date absolue de type *Time*.

Les Figure 4, Figure 5 et Figure 6 décrivent trois extraits de machine à états-transitions illustrant l'utilisation possible des évènements temporels de *UML*.

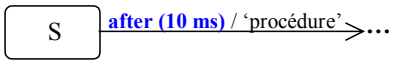


Figure 4. L'événement temporel est généré 10 ms après la date d'entrée dans l'état S.

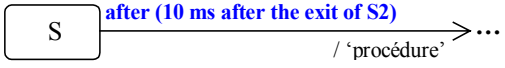


Figure 5. L'événement temporel est généré 10 ms après la date de sortie de l'état S2.

Figure 6. L'événement temporel est généré le 1ier Janvier 2000 à 0h00.

Dans les trois cas, lorsque le temporisateur armé arrive à échéance, il génère un évènement qui est stocké comme tout autre évènement dans la file d'attente associée à la machine d'états. Si celle-ci est dans l'état ***S*** au moment où l'événement temporel est sélectionné, l'événement est consommé et la transition est tirée. Dans le cas contraire, l'évènement temporel est perdu. Contrairement aux autres événements, les événements temporels ne peuvent être conservés (***defered***).

La date d'occurrence d'un évènement temporel est la même que sa date de réception. Cependant, il faut noter qu'il peut exister un délai variable et difficilement mesurable entre la date de réception d'un évènement temporel et le moment auquel il est pris en compte par l'objet récepteur. En effet, ce délai est la conséquence de la politique de stockage et de traitement mise en œuvre par chaque méthode pour gérer la file des évènements reçus [5-7].

2.4.3. Temps et interactions

La Figure 7 décrit un diagramme de séquence illustrant l'utilisation des notations du temps (observations et contraintes de temps). Un objet de type *User* envoie un message nommé *Code* dont mesure la durée (*d*=***duration***). Une instance de ACSystem envoie alors en retour deux messages : *CardOut* qui doit durer entre 0 et 13 unité de temps ; et *OK* qui doit être reçu entre d et 3xd unité de temps après l'émission du message *Code*. On remarquera que l'on observe également le temps *t* comme étant l'instant d'émission du message *CardOut*. Finalement, ce message doit également être reçu au plus tard 3 unités de temps après sa date d'émission.

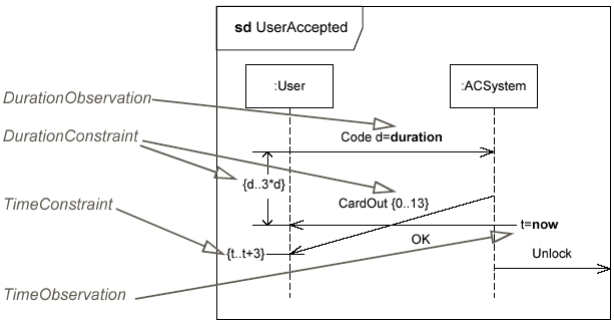


Figure 7. Modélisation du temps dans les diagrammes de séquence.

3. Les extensions pour le temps-réel

Les extensions courantes et normées de UML pour le TR sont essentiellement dédiées à l'analyse des propriétés TR de modèle.

3.1. Annoter pour analyser

Valider les propriétés non-fonctionnelles, en particulier temps-réel, d'une application est une activité aussi importante dans ce domaine que de décrire l'application elle-même. Pour ce faire, il existe un grand nombre de résultats issus principalement de quatre communautés de recherche : l'analyse d'ordonnançabilité, l'analyse de performance, la simulation et la vérification de propriétés temps-réel. Dans chacun de ces cas, les propositions reposent sur des formalismes spécifiques et propriétaires à partir desquels sont définies des techniques d'analyse spécifique : RMA, théorie des files d'attente… Dans le contexte d'une approche dirigée par les modèles, et plus particulièrement par des modèles UML2, il est donc nécessaire de définir des transformations de modèle permettant de traduire les modèles d'une application vers le formalisme d'entrée d'un espace technologique adapté à l'analyse (Figure 8).

Les technologies mises en œuvre pour l'analyse sont souvent très mal perçues par les utilisateurs. Les raisons de cette mauvaise réputation sont souvent dues au formalisme d'entrée et/ou aux théories mathématiques requises. Pour palier à cela, la solution, désormais classique à ce problème, est d'utiliser ce type de technologie d'analyse de façon transparente pour l'utilisateur. Il est alors nécessaire de définir des transformations retours de modèle, i.e. des passages de l'espace des technologies d'analyse choisies vers UML. Il s'agit de convertir les résultats de l'analyse en terme de modèle UML ou d'annotations ajoutées au modèle analysé au départ.

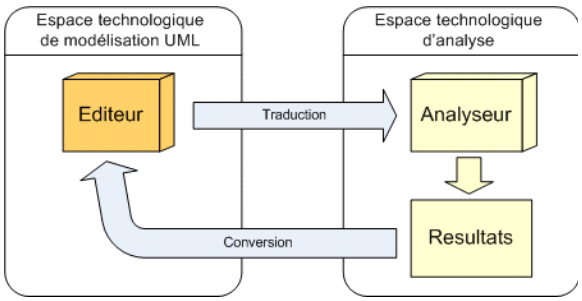


Figure 8. Lien entre modélisation et analyse.

Enfin, quelque soit la technique d'analyse visée, elle requière généralement des informations supplémentaires à celles disponibles dans un modèle de spécification ou de conception classique. On a réellement besoin d'une vue spécifique dédiée à l'analyse, c.a.d. un modèle disposant de toute l'information qualitative et quantitative requise pour la technologie d'analyse visée. Par exemple, pour une analyse d'ordonnançabilité, on a besoin du modèle de tâches et pour chaque tâche d'un certain nombre de propriétés temps-réel telles que l'échéance ou le temps d'exécution au pire cas.

Ce besoin d'annotation de modèle spécifiques à l'analyse est en partie satisfait par le profil UML SPT (*Schedulability , Performance and Time*) [8].

3.2. Le profil SPT

Afin de couvrir les besoins d'analyse TR des modèles UML, l'OMG propose le profil UML SPT. L'intérêt principal de cette norme est qu'elle fournit les moyens d'annoter un modèle avec des caractéristiques de QdS (« Qualité de Service ») permettant d'effectuer des analyses quantitatives. Il est alors possible de vérifier et valider des propriétés extra fonctionnelles comme les temps de réponse ou les tailles des files d'attente en se basant sur des données comme les échéances, les pires temps d'exécution (WCET) ou les politiques d'ordonnancement.

Le profil SPT se compose de trois paquetages principaux (Figure 9). Le premier paquetage, nommé GRMF (*General Resource Modeling Framework*), définit un cadre générique pour supporter n'importe quel type d'analyse de modèle. Il est lui-même constitué de trois sous-paquetages dédiés à la modélisation des ressources (GRM pour *General Resource Modeling*), de la concurrence (GCM pour *General Concurrency Modeling*) et du temps (GTM pour *General Time Modeling*). Ces paquetages spécifient respectivement les moyens de modéliser la QdS des ressources en considérant les propriétés physiques du logiciel et du matériel support d'exécution d'une application temps-réel, de définir le concept d'unité concurrente et de représenter les concepts liés au temps.

Le second grand paquetage du SPT spécialise le paquetage précédant en réifiant les concepts génériques de façon à d'une part définir les moyens nécessaires à une analyse d'ordonnançabilité et d'autre part à une analyse de performance. Le dernier paquetage (qui n'est pas détaillé ici faute de place) est dédié à la description de la plateforme d'exécution *Real-TimeCORBA*.

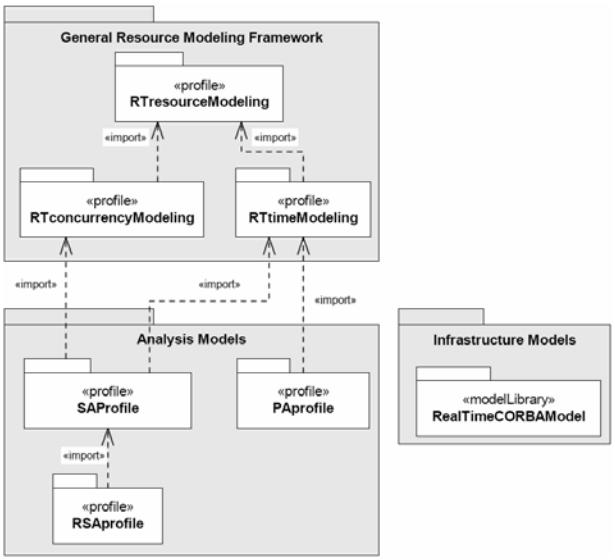


Figure 9. Structure du profil SPT (extrait de [8]).

Le reste de cette section décrit les détails des constituants du GRMF, ainsi que le paquetage d'analyse d'ordonnançabilité.

3.2.1. Le profil de modélisation des ressources TR

Sous la forme d'un diagramme de paquetage de UML2, la Figure 10 décrit, le modèle de domaine du cadre générique fondant le SPT, le profil *RTresourceModeling*.

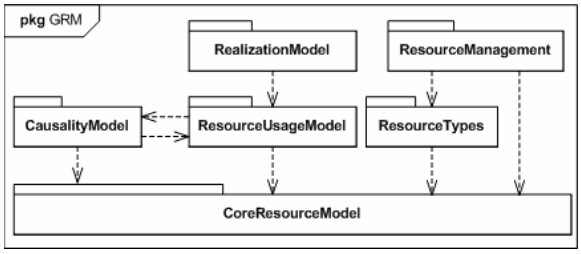


Figure 10. Architecture du GRM.

L'objectif principal des méthodes d'analyse est de répondre à la question : « est-ce que l'offre est suffisante pour réponde à la demande ? » [9]. Par conséquent, le paradigme générique choisi par le SPT pour analyser quantitativement une application temps-réel et défini dans le modèle des ressources principales (*CoreResourceModel*) est le paradigme client-serveur : un client spécifie un ensemble d'exigences non-fonctionnelles (QdS requises) et un serveur fournit des services caractérisés par des exigences non-fonctionnelles offertes (QdS offertes). La relation entre les serveurs et les clients définit par un contrat de QdS.

En dépit du fait que le modèle des ressources principales introduise le concept de QdS, c'est en pratique sous la forme d'attributs additionnels que sont modélisés les caractéristiques non-fonctionnelles d'une ressource (ou d'un service d'une ressource) et non en terme d'instance de QdS associée à cette même ressource (ou service de ressource). C'est parce que les modèles ainsi annotés sont concis que se justifie ce choix. Par exemple, une action possèdera des attributs spécifiant sa date de début, sa date de fin, sa durée et son temps d'exécution au pire cas.

Un point important du GRM est son modèle de causalité défini dans le paquetage éponyme. Le modèle de causalité du SPT est la base de toutes les descriptions dynamiques associées au profil. Il traduit les chaînes de causes à effets du comportement des instances d'une application. Il est basé sur le concept d'occurrence d'évènement. Le concept d'évènement est à prendre au sens de UML, c.a.d. un type de changement d'état d'un système. Une occurrence d'évènement est une instance d'évènement. Deux types d'occurrence d'évènement sont particulièrement intéressants : la génération et la réception d'un stimulus. Le stimulus représente un échange de message entre deux objets communicants, l'objet appelant et l'objet appelé. L'objet appelant exécute une action qui entraîne l'apparition d'une occurrence d'événement de type génération d'évènement et la création d'un stimulus. Celui-ci est éventuellement perçu par l'objet appelé sous la forme d'une occurrence d'événement de type réception de stimulus. En réaction, l'objet appelé peut alors soit exécuter l'une de ses méthodes, soir tirer une transition si son comportement est décrit par une machine d'états. Dans les deux cas, cela mène à l'exécution d'un scénario, c.a.d. une suite ordonnée d'actions qui, à leur tour, peuvent générer des stimuli, et ainsi de suite jusqu'à ce qu'il n'y ait plus de stimuli générés. On notera l'existence de deux autres types d'occurrence d'événement liés au concept de scénario : l'évènement de début d'un scénario et l'évènement de fin d'un scénario.

Le GRM décrit principalement les concepts génériques du SPT vus du côté du serveur (c.a.d. les ressources). L'aspect client est décrit dans le modèle d'utilisation des ressources (paquetage *ResourceUsageModel*) au travers du concept d'usage d'une ressource. Ce dernier décrit comment un ensemble de clients utilise un ensemble de ressources et leurs services associés. On distingue les usages statiques des usages dynamiques. En effet, dans la plupart des techniques d'analyse, il est suffisant de spécifier quels sont les ressources et services sous-jacents utilisés sans entrer dans les détails de l'usage. Dans ce cas, un client spécifie uniquement les relations structurelles qu'il entretient avec les ressources. S'il s'agit d'un type (classe, composant…), il peut spécifier par exemple une association ou une dépendance de type usage. S'il s'agit d'une instance (objet, instance de composant…), il peut spécifier un lien vers les instances ressources ou les services de ressource.

En revanche, une technique d'analyse peut également nécessiter de détailler l'usage d'une ressource, à savoir les aspects liés à l'ordre et au temps. Le cas échéant, il est possible de passer par le concept d'usage dynamique pour décrire plus finement la relation du client vis-à-vis de ses ressources. Un usage dynamique est alors considéré

comme un scénario, et plus particulièrement, comme une suite ordonnée d'exécutions d'action. Cette suite se conforme au modèle prédécesseurs-successeurs, avec la possibilité d'avoir des prédécesseurs et des successeurs multiples et concurrents.

L'usage d'une ressource est lié au fonctionnement interne d'un système qui résulte d'un événement extérieur appelé la demande d'usage (*UsageDemand*). Séparer l'usage de sa demande introduit une plus grande modularité dans la modélisation. Il est ainsi possible d'attacher à différentes demandes d'usage un usage de ressource identique et réciproquement. Les valeurs des caractéristiques de QdS associées à une demande d'usage spécifient les niveaux de QdS requise du système pour un usage donné. Enfin, le modèle d'utilisation des ressources définit le concept de contexte d'analyse (*AnalysisContext*). Le contexte d'analyse est l'ensemble des instances des ressources impliquées dans une analyse, ainsi que l'ensemble des usages de ressources et de leurs demandes associées. Le contexte d'analyse permet de spécifier le point de départ d'une analyse par rapport à un modèle cible. A partir de cette information, un outil d'analyse est capable d'explorer un modèle et d'en extraire toute l'information pertinente pour l'analyse.

Au travers du modèle des types de ressource (*ResourceTypes*), le GRM propose trois taxonomies possibles des ressources :

1. En fonction de leur but, on considère : les ressources de type processeur (physique ou virtuel) qui désignent les entités capables de stocker et d'exécuter le code d'un programme ; les ressources de communications qui permettent aux autres ressources d'échanger de l'information ; les périphériques qui regroupent les ressources qui ne sont ni des processeur, ni des média de communication.
2. En fonction de leur nature, active ou passive : une ressource active est capable de générer des stimuli de son propre chef sans être sollicitée par un stimulus explicit extérieur (ex. une tâche d'un système d'exploitation, un composant matériel…). Une ressource qui n'est pas active est alors dite passive.
3. En fonction de leur protection associée : les instances de ressources protégées proposent au moins une instance de service d'exclusivité. Cette dernière restreint l'accès concurrent à la ressource en spécifiant une politique de contrôle d'accès. Une ressource non protégée est une ressource qui ne propose aucun mécanisme de protection d'accès.

Le modèle de gestion des ressources (*ResourceManagement*) définit le cadre de modélisation des différents types de services de gestion des ressources communément trouvés dans les systèmes d'exploitation. Ce modèle identifie deux concepts en particulier : le courtier des ressources et le gestionnaire des ressources. Le courtier est en charge d'allouer et de libérer un ensemble d'instances de ressources utilisées par un client en fonction d'une politique d'accès aux ressources. Quant au gestionnaire, il est en charge de la création des ressources et en général de leur cycle de vie,. La gestion d'un ensemble d'instances de ressources obéit à une politique donnée de contrôle des ressources. En général, dans les systèmes d'exploitation, courtier et gestionnaire sont une même et unique entité pour un type de ressource.

Le dernier modèle du GRM concerne le déploiement d'une application (paquetage *RealizationModel*). Les architectures en couche sont l'une des techniques les plus usitée en génie logiciel pour structurer une application. SPT distingue deux types d'architecture en couche. Le premier type consiste à considérer que le modèle d'une application est une suite continue de couches construites par des raffinements successifs. Une couche d'un niveau abstrait donné est raffinée en une couche de plus bas niveau (c.a.d. contenant plus de détails en terme de réalisation logiciel). On parle ici d'architecture basée couche de raffinement. Par exemple, un modèle métier peut être raffiné en un modèle C++. Le second type d'architecture en couche, le plus classique, consiste à considérer qu'une couche de modélisation abstraite se réalise au travers des services proposés par une couche de modélisation concrète. On parle alors d'« architecture basée couche de réalisation ». Par exemple, une application temps-réel s'appuie souvent sur un support d'exécution de type système d'exploitation pour les aspects multi-tâches. Les entités du modèle le plus abstrait sont qualifiées d'éléments du modèle logique, alors que les entités du modèle le plus concret sont, elles, qualifiées d'éléments du modèle d'ingénierie. Le mot ingénierie a été choisi ici parce que la couche la plus basse est souvent plus orientée technologie que la couche logique qui manipule des éléments de modélisation plus abstrait, (ou encore orientés métier).

De façon sous-jacente à ce second style architectural, se pose le problème du déploiement. En effet, la question qui se pose dans ce cas est de savoir comment les éléments du modèle logique sont réalisés par les éléments du modèle d'ingénierie. Par rapport à la terminologie de UML, on parle de déploiement du modèle logique sur le modèle d'ingénierie.

Ce concept de réalisation du niveau logique par le niveau d'ingénierie est un cas particulier du paradigme client-serveur vu précédemment. Les éléments du modèle logique sont vus comme les clients des instances de ressources (les serveurs), c.a.d. des éléments du modèle d'ingénierie. De ce fait, un élément du modèle logique peut spécifier des propriétés de QdS requise, alors qu'un élément du modèle d'ingénierie peut spécifier des propriétés de QdS offerte.

Tout ce qui vient d'être dit définit en fait ce qu'on appelle le point de vue du modèle de domaine du SPT pour le GRM, c'est-à-dire qu'il s'agit d'un ensemble de concepts de base que l'on juge utile pour annoter un

modèle en vue d'une analyse quantitative. Si maintenant on vise à annoter des modèles UML, il faut projeter ce modèle de domaine sur le méta-modèle de UML et définir ainsi un profil UML. En effet, UML définit le concept de profil comme moyen d'extension du langage à un domaine particulier. Définir un profil passe entre autre par décrire des stéréotypes et des propriétés de stéréotypes. Un stéréotype est un moyen d'étendre un ou plusieurs éléments du langage UML, lui-même défini par un méta-modèle.

La plus part des éléments du modèle de domaine du GRM reste abstrait, c.a.d. que l'on ne définit pas de stéréotypes UML pour les manipuler en modélisation. Ils sont réifiés dans le SPT par les sous-profils d'analyse spécifique comme celui de l'analyse d'ordonnançabilité détaillé ci-après. Le point de vue UML du GRM concerne le concept de déploiement et les services d'acquisition et de libération d'une ressource introduite dans le modèle de gestion des ressources. Pour des raisons de place, nous ne détaillerons ici que ce qui est relatif au déploiement. Pour le reste, le lecteur est invité à lire la section 3.2.2.3 de la norme [8].

UMl propose le concept de raffinement au travers du concept d'abstraction stéréotypé par « refine ». Les trois types de réalisations introduites par le SPT et décrites ci-après seront ainsi des spécialisations du stéréotype « refine » de UML :

1. « GRMcode » décrit l'association du code fournit par un client et d'un composant ou d'un nœud UML.
2. « GRMdeploys » décrit le fait que des instances d'un ou plusieurs clients sont déployées sur un serveur.
3. « GRMrequires » spécialise le déploiement précédent. Dans cette relation, les clients sont alors considérés comme une spécification générique du minimum requis pour le déploiement sur le serveur.

Pour « GRMcode » et « GRMdeploys », la QdS atteignable par un client est contrainte par les caractéristiques de QdS offertes par le serveur. La sémantique précise de la contrainte est fonction de la nature de la QdS et elle doit être précisée à l'utilisation.

De plus, on peut préciser le mode d'une association :

1. Mode inclusif, si la fonctionnalité d'un client utilise l'ensemble des services du ou des serveurs.
2. Mode exclusif statique, si l'élément client est réalisé par un et un seul client et que cela ne change pas durant l'exécution du client.
3. Mode exclusif dynamique, si l'association peut changer au cours de l'exécution du client.

Pour *GRMdeploys*, la nature du raffinement peut varier en fonction du type de client et de serveur. SPT liste trois types d'interactions possibles :

1. Synchrone si clients et serveurs sont des entités logicielles et qu'ils sont en interaction uniquement au travers d'appel d'opérations.
2. Asynchrone si clients et serveurs sont des entités logicielles et qu'ils sont en interaction asynchrone uniquement (c.a.d. appel d'opération asynchrone ou communication par signal).
3. De remplacement si clients et serveurs sont respectivement logiciels et matériels. Dans ce cas, le comportement du logiciel devient celui du matériel. Ce déploiement ressemble alors à un raffinement à la différence près que client et serveur sont des entités différentes.

Le cas où clients et serveurs sont matériels est considéré dans le SPT comme un raffinement plutôt que comme un déploiement.

3.2.2. Le profil de modélisation du temps

La Figure 11 décrit sous la forme d'un diagramme de paquetage de UML2, le modèle de domaine du cadre générique relatif au temps : GTM (*General Time Modelling*).

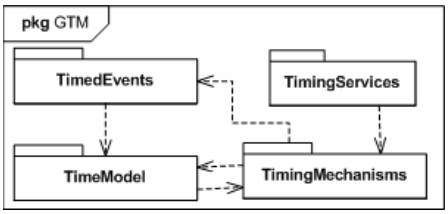


Figure 11. Architecture du GTM.

L'objet de ce modèle est de définir des concepts relatifs à la mesure du temps, comme des durées ou des temps d'horloges. Il exclut donc les modèles de temps logique.

Le modèle de temps du SPT s'appuie principalement sur le concept de temps physique considéré comme un ensemble continu et non borné des instants du temps physique (Figure 12). Cet ensemble est complètement ordonné et dense[2]. Le concept de temps physique est concrètement manipulable dans les modèles au travers de deux concepts : *TimeValue* et *TimeInterval*. Une valeur de temps (*TimeValue*) peut prendre deux formes : les valeurs de temps discret (représentables par un entier par exemple), et les valeurs de temps dense (représentables par un réel par exemple). Un intervalle de temps est une spécialisation du concept de valeur de temps. Un intervalle de temps peut donc être de temps discret ou de temps dense. De plus, un intervalle de temps possède deux propriétés supplémentaires : un temps de début et un temps de fin.

[2] Denses signifie qu'entre deux instants, on trouve toujours un instant.

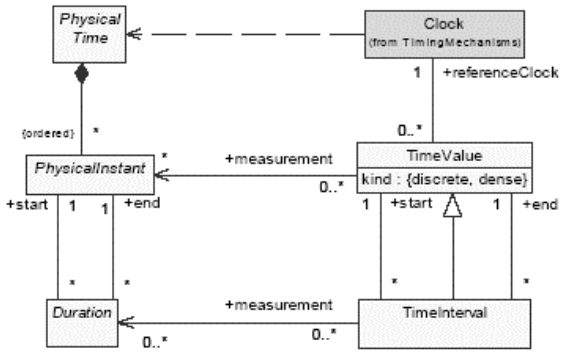


Figure 12. Concepts de base du modèle de temps du SPT.

Pour mesurer le temps, les deux mécanismes les plus couramment utilisés dans les systèmes d'exploitation temps réel sont les temporisateurs et les horloges. Les temporisateurs sont des mécanismes capables de générer un événement de type expiration du temps (*timeout*) quand un instant de temps donné arrive. Ce temps peut être absolu (une date) ou relatif (un intervalle entre une date de référence et une échéance). Un temporisateur peut être périodique. Une horloge est un mécanisme qui périodiquement génère un évènement appelé le tic d'horloge. Les mécanismes temporels possèdent des propriétés particulières comme une valeur courante, une valeur maximale, une dérive et offrent des services du type *setTime*, *getTime* ou *reset*. Enfin, ces deux mécanismes temporels sont aussi des types de ressource. Ils peuvent donc associer des niveaux de QdS offerte à leurs services associés.

SPT offre la possibilité d'avoir des stimuli étiquetés temporellement : *TimedStimulus*. Un stimulus temporisé possède au moins une étiquette spécifiant sa date d'émission. Si on n'est pas dans un système disposant d'une référence globale du temps, il est alors possible de définir plusieurs étiquettes temporelles référençant chacune des horloges différentes. Un stimulus temporisé peut également posséder une ou plusieurs étiquettes temporelles spécifiant la ou les dates de réception.

SPT définit deux types particuliers de stimulus : *Timeout* et *ClockInterrupt*. Ceux-ci sont des stimuli issus respectivement de l'arrivée à échéance d'un temporisateur et du tic d'une horloge.

Deux autres concepts sont également utiles pour modéliser des applications temps-réel : le concept d'évènement temporisé (*TimedEvent*), c.a.d. possédant une étiquette de temps (date d'émission), et le concept d'action temporisée (*TimedAction*). Une action temporisée a une durée de type intervalle de temps, un début et une fin. Une action nommée *Delay* spécialise *TimedAction* pour modéliser une action particulière dont l'exécution a pour effet de retarder l'exécution d'un intervalle de temps donné.

Tous les concepts que l'on vient de voir et qui sont définis dans le modèle de domaine du SPT pour le temps sont projetés en UML et donc transcrits en terme de stéréotypes et de propriétés associées. On ne liste pas ici l'ensemble des stéréotypes définis mais on donne juste un exemple. La Figure 13 spécifie une classe *Date* de type valeur de temps (usage du stéréotype « RTtime ») et qui référence une horloge spécifique portant l'identifiant *sysMaster*. L'objet, *myMasterClock*, instance de la classe *AClck* est déclarée être un objet de type *Clock* au sens SPT (« stéréotype « RTclock ») et son identifiant est *sysMaster* (défini dans la valeur étiquetée associé *RTclockId*).

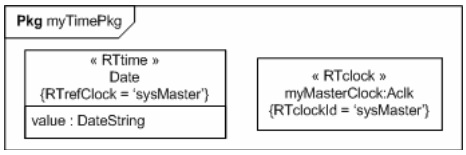


Figure 13. Exemple d'usage de stéréotypes du GTM.

Le profil SPT définit également quelques types de base spécifiques au temps comme *RTtimeValue*. Pour ce faire, il s'appuie sur TVL (« Tag Value Language »), un langage proposé par SPT pour décrire de façon précise les valeurs des valeurs étiquetées. TVL est décrit dans l'annexe A de [8].

3.2.3. Le profil de modélisation de la concurrence

Le modèle général de concurrence du SPT définit comme concept de base l'unité de concurrence (*ConcurrentUnit*), qui est une entité active qui peut s'exécuter en parallèle d'autres entités concurrentes. Après sa création, chaque entité concurrente commence à exécuter une méthode particulière appelée le scénario principal (*main scenario*). Le scénario principal s'exécute tant que l'entité concurrente existe. Au cours de son exécution, le scénario principal peut explicitement appeler à des actions de réception pour accepter les stimuli qui lui sont envoyés. La réception d'un stimulus entraîne alors l'exécution du service approprié. Pendant cette exécution, soit le scénario principal est bloqué jusqu'à achèvement du service (sémantique de « run-to-completion »), soit il peut continuer à s'exécuter en parallèle de l'exécution du service requis.

Une unité concurrente dispose d'une ou plusieurs files d'attente lui permettant de stoker les stimuli reçus alors qu'elle n'était pas en état de les traiter.

Dans le contexte d'échange entre deux unités concurrentes, deux options impactent le modèle de causalité :

1. Du coté du client, si la communication est asynchrone, le client peut continuer à s'exécuter. En revanche, si la communication est synchrone, le client est bloqué jusqu'à ce que le serveur ait envoyé une réponse.
2. Du coté du serveur, une requête peut soit être traitée immédiatement, soit déférée. Si le

traitement doit être immédiat, il se fait soit dans le contexte d'un fil d'exécution (*thread*) existant (option dite à distance, i.e. *remote*), soit via un fil d'exécution créé par le serveur à la réception de la requête (option dite locale).

Du point de vue UML, les unités concurrentes d'un modèle sont marquées du stéréotype « RTconcurrent ». Il peut s'agir de classes ou d'instances. S'il s'agit d'une classe, cela signifie que toutes les instances de la classe seront des unités concurrentes. Le scénario principal est spécifié au travers d'une valeur étiquetée attachée à la classe et donnant le nom de l'opération de la classe correspondant au « main ».

3.2.4. L'analyse d'ordonnançabilité avec SPT

Le paquetage de modélisation de l'ordonnançabilité fournit un ensemble minimal d'annotations nécessaires à la description d'un modèle en vue d'une analyse de son ordonnançabilité. Cet ensemble permet de pratiquer des analyses basiques, statiques ou dynamiques. Chaque utilisateur voulant appliquer des techniques plus sophistiquées est invité à fournir un profil étendant le profil d'analyse d'ordonnancement du SPT.

La Figure 14 décrit les concepts de bases nécessaires pour décrire un modèle d'exécution pour une analyse d'ordonnançabilité.

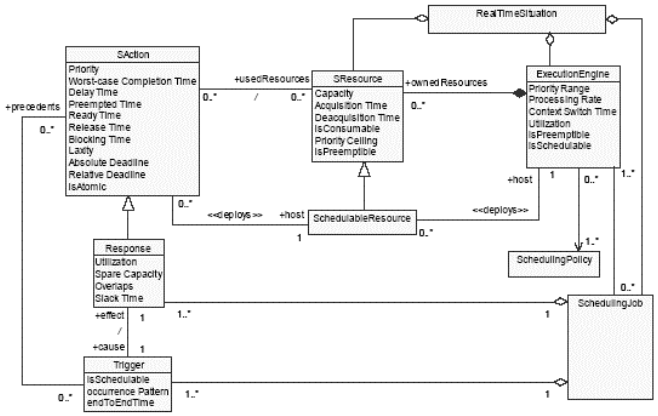


Figure 14. Modèle principal d'ordonnançabilité.

Le concept introduit par le SPT pour supporter la notion de tâche est le concept nommé *SchedulableResource*[3]. Il s'agit en effet des entités utilisées pour exécuter les actions. Les tâches sont alors déployées sur un moteur d'exécution (*ExecutionEngine*), et elles exécutent des actions (*SAction*) qui ont des propriétés TR comme leur temps d'exécution au pire cas.

Le moteur d'exécution qui exécute un *scheduling job* peut être une CPU, un système d'exploitation ou une machine virtuelle.

La Figure 15 décrit les concepts de l'infrastructure et les mécanismes relatifs à l'analyse d'ordonnançabilité.

[3] Par la suite, le mot tâche sera utilisé pour traduire le concept nommé « SchedulableResource ».

D'une part, l'ordonnanceur affecte les tâches aux ressources d'exécution en fonction d'algorithmes d'ordonnancement et, d'autre part, il arbitre l'usage des autres types de ressources requises par les tâches en fonction de politiques spécifiques d'arbitrage des ressources.

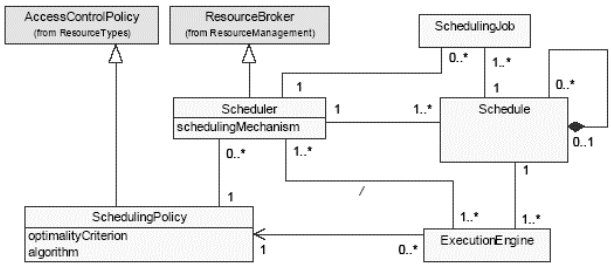


Figure 15. Concepts de l'infrastructure et mécanismes relatifs à l'analyse d'ordonnançabilité.

3.2.5. Exemple d'utilisation

C'est un système de régulation de vitesse qui servira ici à donner un exemple d'utilisation du SPT décrit précédemment. La Figure 16 décrit l'architecture du système, obtenu par application de la méthodologie Accord|$_{UML}$.

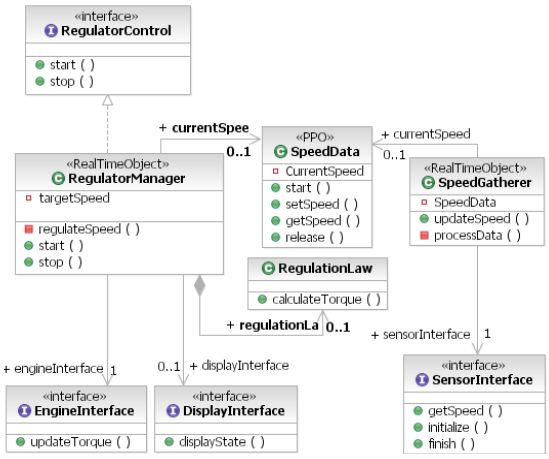


Figure 16. Architecture du régulateur de vitesse.

Afin de pouvoir faire une analyse de ce modèle, il faut définir un modèle d'instances de l'application (c.a.d. déterminer quels sont les objets qui s'exécutent). Dans le système de l'exemple, on considèrera qu'il existe une seule instance de chaque classe comme décrit à la Figure 17.

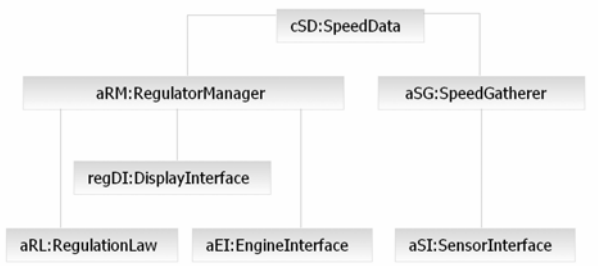


Figure 17. Modèle d'instance du régulateur de vitesse.

Sur le modèle de la Figure 16, on peut remarquer que les deux classes *RegulatorManager* et *SpeedGatherer* portent le stéréotype « RealTimeObject ». De ce fait, les instances de ces classes peuvent s'exécuter en concurrence. Plus précisément, cela signifie que les appels aux opérations *start* et *stop* des instances de *RegulatorManager* et les appels à l'opération *updateSpeed* de *SpeedGatherer* s'exécuteront avec leur propre unité de concurrence, c.a.d. leur tâche.

D'un point de vue modèle d'exécution, le système s'appuiera donc sur trois tâches : une première pour la régulation de la vitesse ($T_{regStart}$), une deuxième pour l'arret de la régulation ($T_{regStop}$) et une troisième pour l'acquisition de la vitesse. (T_{acq}) Le point d'entrée de *Treg* est l'opération *regulateSpeed* de la classe *RegulationManager* et le point d'entrée de *Tacq* est l'opération *updateSpeed* de la classe *SpeedGatherer*.

Pour l'analyse d'ordonnançabilité, il est important d'identifier ce que l'on nomme des situations stables (nommées situations de temps réel « *RealTimeSituation* » dans SPT) afin de générer un modèle utilisable par les différents outils d'analyse.

Pour le système de régulation, on peut identifier trois situations stables correspondant aux trois tâches précédemment identifiées. Le diagramme de la

Figure 18 décrit le cas de la tâche d'acquisition de la vitesse et précise toutes les informations qualitatives nécessaires à une analyse d'ordonnaçabilité.

L'événement qui déclenche le scénario est stéréotypé « *SAtrigger* », et on caractérise sa QdS par les lois d'activation (périodique, apériodique, etc.) et le temps d'inter-arrivée entre deux évènements. La chaîne d'actions d'exécution en réponse à ce déclencheur est stéréotypée comme « *SAresponse* » et est annotée avec l'information d'échéance («*SAabsDeadline* »). Enfin, chaque morceau d'exécution atomique (stéréotypé « *SAaction* ») de la réponse est annoté avec des métriques, comme le temps d'exécution de pire cas (« *SAworstCase* »).

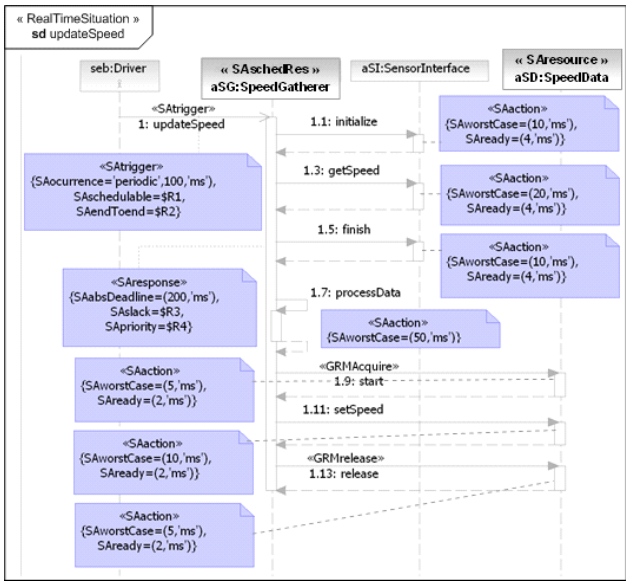


Figure 18. Exemple d'annotation TR d'une transaction.

3.3. Le profil QoS&FT

Le profil QoS&FT (*Quality of Services & fault Tolerance*) a été adopté en septembre 2004 comme spécification finale par l'OMG [10]. Ce profil est basé sur la norme UML2 et il est en conformité avec le profil SPT précédemment décrit.

3.3.1. Principe

Le profil QoS&FT permet de prendre en compte les différentes caractéristiques de qualité de service et de tolérance aux fautes d'un système. Ces deux notions sont traitées de façon disjointe dans le profil.

Les exemples de caractéristiques de qualité sont la périodicité, l'irrégularité, les limites, la gigue, les échéances ou encore le type d'une exigence (dure, moyenne, souple). Les niveaux de qualité expriment des degrés de satisfaction d'une propriété non fonctionnelle. Ils sont exprimés au travers de contrats qui sont eux-mêmes associés aux caractéristiques de QdS.

Dans ce profil, un méta-modèle est spécifié permettant de définir un concept abstrait d'un langage de modélisation supportant des concepts de modélisation de QdS. On ne définit pas de syntaxe concrète à ce niveau.

Le méta-modèle dédié à la QdS est composé de trois paquetages. Le premier paquetage spécifie les caractéristiques de QdS. Il donne les bases permettant de définir les contraintes de QdS dans un second paquetage. Ce second paquetage peut être ensuite utilisé dans les contrats qui servent finalement à définir les niveaux de QdS, concept introduit dans le troisième et dernier paquetage.

Pour situer ce profil par rapport à SPT, les définitions des caractéristiques et des valeurs de QdS dans le profil QoS & FT sont des spécialisations des définitions de ces

notions introduites dans le SPT. Les contrats sont des relations exprimées entre les QdS offertes et requises. Les relations entre les ressources et la QdS sont plus spécialement adressées dans le profil SPT.

L'intégration de ces notions au sein de UML se fait par la définition de profils UML associés aux différents paquetages décrits précédemment.

On définit également dans ce profil comment traiter et décrire la gestion des risques et la tolérance aux fautes. Pour cette dernière, la solution proposée s'appuie principalement sur des systèmes complexes et généralement distribués avec des exigences de robustesse. La façon de traiter la tolérance aux fautes dans ce profil est uniquement liée à la robustesse d'un système, c'est-à-dire à la propriété de continuité des services d'un système. Les bases de ce profil sur la tolérance aux fautes sont la réplication des objets et des composants et la détection des fautes.

La toute récente adoption de ce profil amènera à regarder attentivement quelles parties de ce profil seront réellement utilisées et comment elles le seront (notamment par rapport à la concurrence entre le profil QoS&FT et le profil SPT).

On trouve dans le profil différentes notions qui sont communes aux profils SPT et QoS&FT notamment dans la partie QdS où les caractéristiques sont également présentes dans le profil SPT. Les travaux effectués dans le cadre de la définition d'un nouveau profil dédié aux systèmes embarqués temps réel (cf. section suivante) visent à unifier dans un même profil les apports des deux profils SPT et QoS&FT.

3.3.2. Exemple d'utilisation

A l'instar du profil SPT, le profil QoS&FT peut être utilisé pour annoter quantitativement des modèles UML, afin de pouvoir procéder à une analyse quantitative des propriétés d'un modèle. La suite de cette section illustrera au travers d'un exemple d'utilisation de QoS&FT pour une analyse d'ordonnançabilité.

QoS&FT n'est pas directement utilisable. Si cela n'a pas déjà était fait, il faut définir un ou plusieurs catalogue sde QdS relativement à l'objectif fixé. Le processus de préparation à l'utilisation de QoS&FT se compose de trois étapes :

1. Un catalogue de QdS est d'abord spécifié sous la forme d'une bibliothèque de classes « templates » où toutes les caractéristiques de QdS et les dimensions de QdS associées sont fixées pour le domaine en question (dans notre cas, le domaine de l'analyse d'ordonnançabilité).
2. Toutes les valeurs des paramètres du catalogue sont alors définies dans un modèle QdS spécifique à l'application. Ce modèle est appelé le modèle de QdS utilisateur. Il définit les classes de caractéristiques de QdS qui seront instanciées dans le modèle final.
3. Finalement, le model UML de l'utilisateur est annoté avec des contraintes de QdS (QdS offerte, QdS requise, ou des contrats de QdS) et les valeurs ajoutées aux instances de classes des caractéristiques de QdS.

Pour notre exemple, nous utilisons le catalogue de QdS spécialisée pour l'ordonnançabilité définie dans [10]. Il reste donc à définir les valeurs des paramètres de ce catalogue dans un modèle spécifique à l'application du régulateur de vitesse. L'unité de temps choisie ici est le milliseconde (ms) (Figure 20).

Figure 20. Extrait du modèle de spécialisation du catalogue de [10] dédié à l'analyse d'ordonnançabilité pour notre exemple.

Une fois qu'on a défini le modèle de QdS adapté à son application, on peut l'utiliser pour représenter les propriétés extra fonctionnelles dans les diagrammes d'instances correspondant au système modélisé. La Figure 21 décrit un exemple d'utilisation de ce modèle de QdS sur le scénario d'acquisition de la vitesse de notre exemple (cf. section 3.2.5). On observera que le style de modélisation préconisé par le profil QoS & FT utilise le langage OCL pour fixer les valeurs numériques associées au modèle.

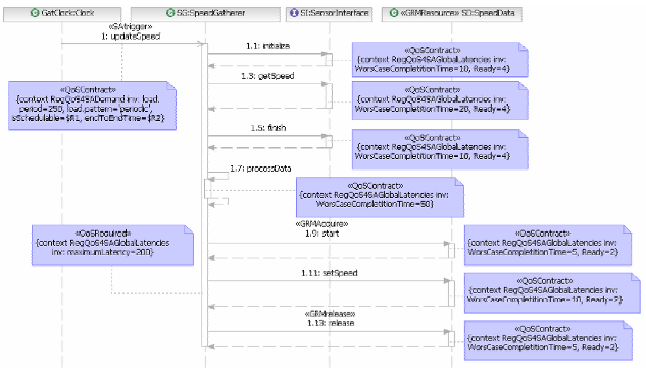


Figure 21. Exemple d'utilisation du catalogue d'ordonnançabilité du QoS&FT adapté à l'exemple de régulation de vitesse.

3.4. De UML vers un espace technologique d'analyse

Une fois le modèle annoté et c,e quelque soit le profil utilisé, l'exploitation d'un modèle en vue d'une analyse se fait usuellement via un outil externe (c.a.d. différent de l'outil de modélisation). Pour ce faire, il est nécessaire de définir une transformation de modèle de UML vers le

formalisme de l'outil utilisé en s'appuyant par exemple sur le langage défini dans [11].

Ce principe de travail a montré qu'il est souvent nécessaire de définir une transformation retour permettant de traduire les résultats fournis par l'outil d'analyse en terme de modèle UML.

Par exemple, dans le cadre du projet Accord|$_{UML}$ développé au CEA, deux transformations ont ainsi été définies (Figure 22).

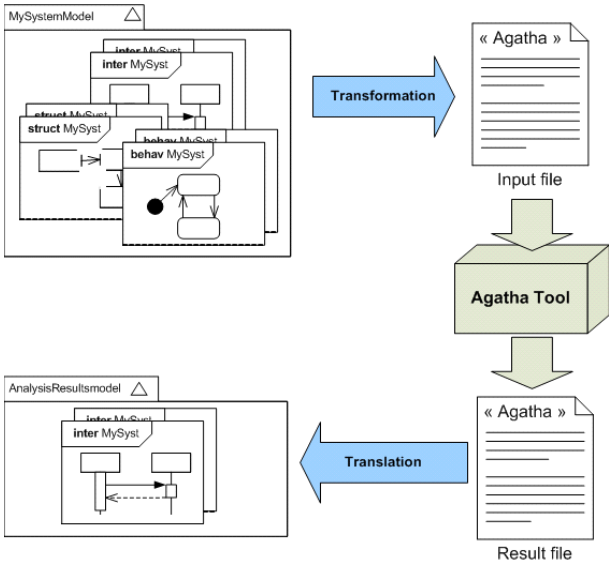


Figure 22. Exemple de liens entre UML et un outil d'analyse externe.

Ces deux transformations permettent de connecter un modèle annoté suivant la méthodologie Accord|$_{UML}$ à un outil d'analyse formel (AGATHA [12]). Ce dernier fournit la capacité à analyser le comportement de spécifications à base d'automates concurrents et communicant par rendez-vous.

4. Conclusions

Depuis le début 2005, l'OMG a lancé un nouveau RFP (« Request For Proposal[4] ») pour le TR/E visant à définir un profil UML pour la modélisation et l'analyse de système TR/E (*Modeling and Analysis of Real-Time and Embedded systems*) [13]. Ce profil remplacera le profil en cours (nommé SPT) et vise à couvrir à la fois le domaine de la modélisation et celui de l'analyse des systèmes temps-réel embarqués (TR/E). La version initiale est attendue pour fin 2005 et la version finale pour fin 2006. Jusqu'ici un seul consortium a été créé pour fournir une réponse (CEA, Thales, INRIA, IBM, Artisan Software, Lookheed, Tripac et université de Carlton).

La Figure 23 décrit l'architecture attendue du profil. Elle est basée sur trois paquetages : le paquetage nommé TCR, définissant les concepts de base nécessaires au domaine du TR/E, à savoir les concepts de temps, de concurrence et de ressource. Dans ce même paquetage, trois modèles de temps particuliers seront définis : synchrone, asynchrone et temps-réel. Le second paquetage nommé RTEM définira les concepts requis pour la modélisation des applications TR/E. Trois besoins importants seront abordés en particulier : la modélisation des plateformes logicielles et matérielles, la définition des modèles de calcul/exécution les plus usités dans le domaine du TR/E, et la modélisation de comportement déterministe. Le dernier paquetage, RTEA, spécifiera les constructions nécessaires à l'analyse de modèles TR/E, et plus particulièrement l'analyse de performance et d'ordonnançabilité.

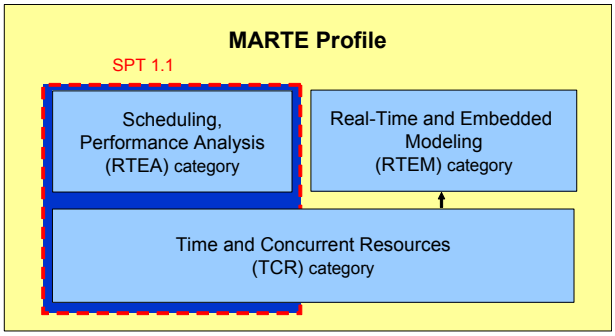


Figure 23. Architecture du profil MARTE.

[4] Un RFP est en quelque sorte un appel d'offre émis par l'OMG pour une nouvelle norme.

References

[1] OMG, "UML2.0 Superstructure Specification", 2004.

[2] S. Gérard, C. Mraidha, F. Terrier, and B. Baudry, "A UML-based concept for high concurrency: the Real-Time Object", presented at The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'2004), T. A. a. I. L. J. Gustafsson, IEEE Computer Society, ISBN 0-7695-2124-X, pp 64-67, Vienna, Austria, 12-14 May 2004 2004.

[3] P. Tessier, S. Gérard, C. Mraidha, F. Terrier, and J.-M. Geib, "A Component-Based Methodology for Embedded System Prototyping", presented at 14th IEEE International Workshop on Rapid System Prototyping (RSP'03), IEEE Computer Society, ISBN 0-7695-1943-1, pp 9-15, San Diego, USA, 9-11 June 2003 2003.

[4] C. Mraidha, S. Gérard, Y. Tanguy, H. Dubois, and R. Schneckenburger, "Action Language Notation for Accord/UML", CEA, DTSI/SOL/LLSP/04-163/HD, 2004.

[5] F. Bause and P. Buchholz, "Protocol Analysis using a timed version of SDL", in *Formal Description Techniques*, J. Quemada, J. Manas, and E. Vasquez, Eds., North Holland, 1991, pp.

[6] S. Leue, "Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-

Based Approach", presented at PSTV, Chapmann & Hall, pp, 95.
[7] U. Hinkel, "SDL and Time - A mysterious Relationship", presented at SDL Forum, pp, 97.
[8] OMG, "UML Profile for Schedulability, Performance, and Time, v1.1", formal/05-01-02, 2005.
[9] B. Selic, "Modeling Quality of Service with UML", in *UML for Real: Design of Embedded Real-Time Systems*, L. Lavagno, G. Martin, and B. Selic, Eds. Boston: Kluwer Academic Publishers, 2003, pp. 369.
[10] OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics & Mechanisms", ptc/04-09-01, 2004.
[11] OMG, "Revised submission for MOF 2.0 Query/View/Transformation RFP", ad/2005-03-02, 2005.
[12] D. Lugato, N. Rapin, and J.-P. Gallois, "Verification and tests generation for SDL industrial specifications with the AGATHA", presented at Workshop on Real-Time Tools, CONCUR'01, pp, 2001.
[13] OMG, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems RFP", realtime/05-02-06, 2005.

The Bossa Framework for Scheduler Development

Gilles Muller
Obasco Group, EMN-INRIA, LINA
44307 Nantes Cedex 3, France
Gilles.Muller@emn.fr

Julia L. Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Abstract

Writing a new scheduler and integrating it into an existing OS is a daunting task, requiring the understanding of multiple low-level kernel mechanisms. Indeed, implementing a new scheduler is outside the expertise of application programmers, even though they are the ones who understand best the scheduling needs of their applications.

We propose a framework, Bossa, to allow application programmers to implement kernel schedulers easily and safely. This framework defines a scheduling interface that is instantiated in a version of the Linux kernel by an Linux expert using an approach based on a variant of Aspect-Oriented Programming. Schedulers are written using a domain-specific language (DSL) that provides high-level scheduling-specific abstractions to simplify the programming of scheduling policies. The use of a DSL both eases scheduler programming and enables verification that a scheduling policy is compatible with OS requirements. We have found that Bossa gives good performance in practice. In this talk, we present the Bossa DSL, its implementation in Linux 2.4, and its use in the context of multimedia applications.

1 Introduction

Process scheduling is an old problem, but there is no single scheduler that is perfect for all applications. Indeed, in the last few years, the emergence of new applications, such as multimedia and real-time applications, and new execution environments, such as embedded systems, has given rise to a host of new scheduling algorithms [2, 3, 5, 7, 14, 18, 19, 21, 22, 23, 24, 25]. Nevertheless, because these algorithms are typically highly specialized, few have been included in commercial operating systems (OSes).

Ideally, when the scheduling behavior required by an application is not available, the application programmer can implement a new scheduler in the target OS. Nevertheless, scheduler programming at the kernel level is a difficult task. First, there is no standard interface for implementing schedulers. Thus, the programmer must identify the parts of the kernel that should be modified and the code that should be written in each case. Because scheduling is affected by all kernel services, this analysis requires a global understanding of the kernel behavior. The analysis is further complicated by the pseudo-parallelism present in the kernel due to interrupts. Second, few debugging tools are available at the kernel level. Indeed, any errors in kernel code are likely to crash the machine, making bugs difficult to track down. Together these issues imply that the kind of expertise required to successfully integrate a new scheduler into an existing OS is outside the scope of application programmers.

Bossa Bossa is a framework to allow application programmers to implement kernel schedulers easily and safely. This framework defines a scheduling interface that is instantiated in a standard OS by an OS expert. Schedulers are written using a *domain-specific language* (DSL) that provides high-level scheduling-specific abstractions to simplify the programming of scheduling policies. To enable compile-time verification that a scheduler interacts correctly with the target kernel, the OS expert configures the DSL compiler with a model of the kernel's scheduling behavior, including information about process state transitions and interrupts. Schedulers can either be compiled with the kernel or dynamically loaded into a scheduling hierarchy. Because Bossa extends a standard OS, applications can continue to use a standard execution environment (drivers, libraries, etc.).

We have implemented Bossa in the Linux 2.4.18 and Linux 2.4.24 kernels. Bossa has been used to implement a variety of scheduling policies, including policies directed towards multimedia applications such as progress-based scheduling [22], policies directed towards real-time systems such as rate monotonic and earliest-deadline first (EDF) [4], and general-purpose policies such as the policy of Linux. Most policies amount to under 200 lines of Bossa code and were implemented in a few hours beyond the time required to understand the scheduling algorithm. Some of these policies were implemented by students with no previous kernel programming experience. Overall, we have found that the use of Bossa allows the scheduler programmer to focus on the features of the policy to be implemented rather than on the details of integrating a new

```
scheduler EDF = {
  process = {
    time period;
    time wcet;
    time current_deadline;
    timer period_timer;
  }
  states = {
    RUNNING running : process;
    READY ready : select queue;
    READY yield : process;
    BLOCKED blocked : queue;
    BLOCKED computation_ended : queue;
    TERMINATED terminated;
  }
  ordering_criteria = { lowest current_deadline }

  handler (event e) {
    On unblock.preemptive {
      if (e.target in blocked) {
        if (!empty(running) && e.target > running) {
          running => ready;
        }
        e.target => ready;
      }
    }
    On bossa.schedule {
      if (empty(ready)) { yield => ready; }
      select() => running;
      if (!empty(yield)) { yield => ready; }
    }
    ...
  }
}
```

Figure 1. EDF scheduler

scheduler into an existing OS.

In the rest of this paper, we describe some features of the Bossa framework. Section 2 introduces the Bossa DSL. Section 3 describes an aspect-oriented approach to integrating Bossa into an OS such as Linux. Section 4 evaluates the performance of our approach and illustrates some applications. Section 5 concludes and describes future work. Bossa has been presented in detail elsewhere [1, 10, 11, 15, 16]. This paper thus provides a brief overview of some of the highlights of the Bossa framework.

2 The Bossa DSL

The goal of the Bossa DSL is to express scheduling policies in a clear, concise and verifiable way. A Bossa scheduling policy includes a set of declarations and a set of handlers for kernel scheduling events. We introduce the language using excerpts of an implementation of an EDF scheduling policy [13], shown in Figure 1, which illustrates most of the language features. The complete implementation is 162 lines of Bossa code. The complete policy and a grammar of the Bossa DSL are available at the Bossa web site.[1]

[1] http://www.emn.fr/x-info/bossa

Declarations The declarations of a scheduling policy define the process attributes, the scheduling states, and the ordering of processes.

The `process` declaration lists the policy-specific attributes associated with each process. Those of the EDF policy are the period and the Worst-Case Execution Time (WCET) supplied by the process, the process's current deadline, and a timer that is used to maintain the period.

The `states` declaration lists the set of process states that are distinguished by the policy. Each state is associated with a state class (`RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state and an implementation as either a process variable (`process`) or a queue (`queue`). The names of the states of the EDF policy are mostly intuitive. For example, the `ready` state is in the `READY` state class, meaning that it contains processes that are able to run. This state is also designated as `select`, meaning that processes are elected from this state. The `computation_ended` state stores processes that have completed their computation within the current period.

The `ordering_criteria` allows the comparison of two processes according to a sequence of criteria based on the values of their attributes. The EDF policy favors the process with the lowest current deadline. The annotation `select` in the declaration of the `ready` state indicates that the associated queue is sorted according to this criterion.

Event handlers Event handlers describe how a policy reacts to scheduling-related events that occur in the kernel. Examples of such events include process blocking and unblocking and the need to elect a new process. We show only the definitions of the handlers `unblock.preemptive` and `bossa.schedule`, which include most of the scheduling-specific language constructs.

Event handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event. The event-handler syntax is based on that of a subset of C, to make the language easy to learn, and provides specific constructs and primitives for manipulating processes and their attributes. These include constructs for testing the state of a process (*exp* `in` *state*), testing whether there is any process in a given state (`empty(` *state*`)`), testing the relative priority of two processes (exp_1 `>` exp_2), and changing the state of a process (*exp* `=>` *state*). The latter operation is the only means of affecting the state of a process, and both removes the process from its current state and adds it to the new one, thus ensuring by construction that every process is always in exactly one state.

An `unblock.preemptive` event occurs when a process unblocks. The EDF handler checks whether the process is actually blocked (`e.target in blocked`), and if so sets the state of the target process to `ready` making it eligible for election. The

handler also checks whether there is a running process (`!empty(running)`) and if so whether the target process has a higher priority than this running process (`e.target > running`). If both tests are satisfied, the state of the running process is set to `ready`, thus causing the process to be preempted.

Process election is performed by the `bossa.schedule` event handler. The kernel invokes this handler only when a new process must be elected and there are some eligible processes. The handler must change the state of some `READY` process to a state in the `RUNNING` state class and is the only handler that is allowed to do so. In the EDF `bossa.schedule` handler the main effect is to elect a process from the state designated as `select` (`ready`, in the case of the EDF policy) using the `select()` primitive, which is defined in terms of the `ordering_criteria`. Nevertheless, because the EDF policy has two `READY` states, `ready` and `yield`, it may occur that the only `READY` process is actually in the `yield` state. In this case, the handler first changes the state of the `yield` process to `ready`. The policy furthermore implements the strategy that a yielding process only defers to other eligible processes until the next context switch. Thus, the handler terminates by changing the state of any process remaining in the `yield` state to `ready`.

The structure of the EDF event handlers is quite simple, and is typical of that of most of the handlers found in Bossa policies. This simplicity, combined with the domain-specific operators and the characterization of process states by state classes, makes it easy for a programmer to understand the algorithm implemented by a Bossa scheduling policy and enables the Bossa DSL compiler to automatically verify that an event handler satisfies OS-specific requirements [10].

Bossa supports both the construction of a single process scheduler, as described above, and the construction of a hierarchy of schedulers. The use of a hierarchy allows multiple process schedulers, each satisfying particular scheduling needs, to coexist in a running OS. In a Bossa hierarchy, the root and interior nodes are *virtual schedulers*, which only manage other schedulers, and the leaf nodes are *process schedulers*, which only manage processes [11].

3 Preparing Linux for Bossa

Figure 2 illustrates the architecture of the Bossa framework. A Bossa scheduling policy is compiled by the Bossa DSL compiler into a component that is implemented as a kernel module. This component exports an interface requesting event notifications from the OS kernel whenever process state changes occur. The component then uses the information received via these event notifications to make scheduling decisions, including the election of a new process.

Preparing an OS kernel for use with Bossa thus requires inserting event notifications throughout the kernel,

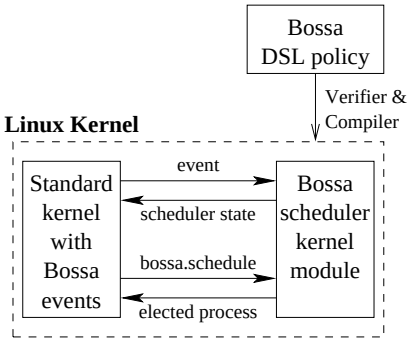


Figure 2. Bossa architecture

wherever process state changes occur, in accordance with the Bossa interface. A standard solution to extending an OS, such as Linux, with a new feature is to perform the integration by hand and to distribute the result as a patch file. Manual integration is, however, tedious and error-prone, and the result is limited to a single version of the OS. To obtain a solution that is both more manageable and more flexible, we have turned for inspiration to aspect-oriented programming (AOP) [8].

AOP is a programming technique that is targeted towards providing a modular implementation of functionalities that crosscut an application. The implementation of such a functionality is isolated in an *aspect*, which contains a collection of code fragments, known as *advice*, and a formal description, known as a *pointcut*, of the positions at which these fragments should be inserted into the target application. To see how AOP can be used for Bossa, we consider the integration of the `unblock.preemptive` event notification into Linux 2.4. In this case, the Bossa functionality completely subsumes the primitive Linux process wakeup function `try_to_wake_up`. Thus, the Bossa aspect contains a pointcut specifying that any call to `try_to_wake_up` should be replaced and an advice specifying that the replacement should call the Bossa unblock event notification function `rts_unblock` with the same set of arguments.

Existing approaches to AOP typically provide pointcut languages that can modify function calls and some kinds of variable references. Because the need for a scheduling interface was not anticipated by the Linux developers, however, the needs of the Bossa interface do not always match up with the structure of the Linux kernel. As an example, we consider the `bossa.schedule` event. The semantics of this event requires that the policy elect a new process and thus coincides roughly with the behavior of the Linux `schedule` function. The Linux `schedule` function, however, does more than elect a new process; it also initiates the context switch and performs some other bookkeeping actions. Thus, we cannot simply replace a call to `schedule` with the `bossa.schedule` event notification; we must instead specify the fragment of the `schedule` definition that the event notification should

replace.

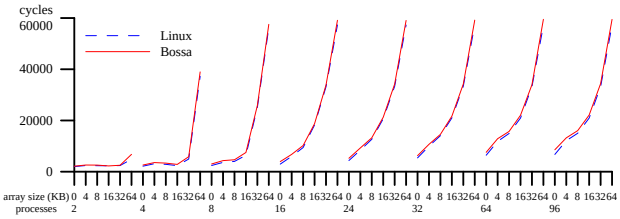
To precisely specify the fragments of Linux code that should be replaced by an event notification, we add features based on temporal logic to the pointcut language. Temporal logic is commonly used to express properties of event sequences, particularly in the context of model checking [6]. Properties include whether an event may eventually occur, or whether it is guaranteed to eventually occur. In the context of specifying properties of programs, we use temporal logic to describe the operations that occur on various paths through a control-flow graph. This use of temporal logic was pioneered by Lacey *et al.*, who use this logic to define rewrite rules that describe common compiler optimizations [9].

In the case of the `bossa.schedule` event notification, we observe that in the Linux `schedule` function, the fragment of code that deals with process election appears after the taking of the runqueue lock and before the releasing of this lock. The Bossa aspect thus specifies that a `bossa.schedule` event notification should replace the maximal code fragment in the `schedule` function in which every instruction satisfies the following property:

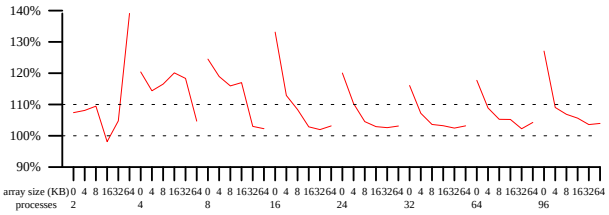
$$\mathsf{AF}\Delta(\texttt{spin\_lock\_irq(\&runqueue\_lock)}) \wedge \mathsf{AF}(\texttt{spin\_unlock\_irq(\&runqueue\_lock)})$$

The operator $\mathsf{AF}\Delta\phi$ matches any code point from which all paths (indicated by A) in a backward direction (indicated by Δ) eventually (indicated by F) reach a point where ϕ is true. In the formula above, ϕ is specified as the code fragment that should be matched. The operator $\mathsf{AF}\phi$ is similar, but considers control-flow in a forward direction. In the formula above, the conjunction of these two operations captures code fragments that are between the taking and releasing of the runqueue lock. These fragments are then as a whole replaced by the `bossa.schedule` event notification. Error checking rules can also be provided in a similar style to check for cases where only some of the control-flow paths satisfy the required property. Using such rules, the Bossa aspect can be applied to multiple versions of the Linux kernel, with the assurance that unexpected code patterns will be detected.

We have implemented an aspect system that provides the above features using the CIL program analysis and transformation framework [17]. Most of the Bossa interface is implemented by an aspect consisting of 23 pairs of pointcuts and corresponding advice. In a few cases, hand modifications are needed *e.g.* in assembly language code and macro definitions, due to limitations of CIL. The aspect code is integrated with the Bossa interface, which thus both describes the interaction between the OS kernel and the Bossa policy, and specifies the means of updating the OS kernel to perform its part of this interaction [15]. We have used the aspect with the standard versions of Linux 2.4.18 and Linux 2.4.24, and with a version of Linux 2.4.24 to which a patch providing high resolution timers had been applied.



(a) Average context-switch overhead (cycles)



(b) Increase in the context-switch overhead when using Bossa. The dotted horizontal lines at 100% and 110% highlight the region in which the overhead of Bossa is below 10%

Figure 3. Comparison of the Bossa implementation of the Linux policy and the native Linux scheduler

4 Performance

The use of Bossa moves scheduling operations from the kernel into a separate module, and thus can have an impact on the context switch time. We use the context switch benchmark `lat_ctx` to measure the impact, and find that it is negligible for real-sized applications. We then consider how Bossa can be used to improve the performance of a video player. All measures were taken using the version of Bossa based on Linux 2.4.18.

Impact on the context-switch overhead Performing a context switch involves electing a new process, saving the register state of the current process, and installing the register state of the elected process. The context-switch overhead also includes the cost of reloading cache and TLB entries as needed during the subsequent execution of the elected process. We measure the cost of these operations using the `lat_ctx` benchmark of the LMBench 2.0.4 benchmark suite.[2] This benchmark passes a token around a ring of processes, triggering a context switch at each step. Each process sums the elements in a local array of a given size to emulate a working set. Varying the size of the array affects the cache and TLB behavior. Figure 3 compares the performance of `lat_ctx` when using the Bossa implementation of the Linux policy to the performance of `lat_ctx` when using the standard Linux scheduler. Measures are grouped first by the array size (0-64KB) and then by the number of processes (2-96). Tests were performed in single-user mode.

[2] `http://www.bitmover.com/lmbench/`

	Min	Max
Linux: Player only	0.005	0.048
EDF: Player only	0.019	0.024
Linux: Player, kernel compilation	0.009	22.683
EDF: Player, kernel compilation	0.017	0.018

Table 1. Distance between Video and Audio

When the overall memory usage (product of the number of processes and the memory usage per process) of `lat_ctx` is below 64KB, the cost of the scheduling policy plays a significant role in the context-switch overhead. Indeed, the use of Bossa increases the overhead by up to 39%, with the worst case being that of 2 processes that manipulate a 64KB array (Figure 3b). When the overall memory usage is above 64-128KB, however, the context-switch overhead increases significantly for both Linux and Bossa. In these cases, the use of Bossa increases the context-switch overhead by only 2-5% as compared to Linux (Figure 3b). While these experiments show some overhead for Bossa, `lat_ctx` represents a worst case, because its computation time is dominated by scheduling and because the memory sizes used are much smaller than those used by real applications running on a general-pupose system.

MPEG video display On a lightly loaded system, a video player can achieve the frame rate required by the video by sleeping for an appropriate time after processing each frame. On a heavily loaded system, the player needs to reserve a portion of CPU time within a fixed interval, to ensure both that it receives adequate access to the CPU and that it receives this access at the appropriate rate.

We consider the use of the video player `mplayer` with a scheduling hierarchy containing a Fixed-priority scheduler at the root, and the Linux 2.4 scheduler and the EDF scheduler of Section 2 at the leaves, as shown in Figure 4. The Linux 2.4 scheduler has lower priority than the EDF scheduler. All processes run on the Linux 2.4 scheduler, except the video player, which runs on the EDF scheduler. We slightly modified `mplayer` to dynamically construct the hierarchy, attach itself to the EDF scheduler, and yield at the end of the processing of each frame. Table 1 shows the behavior of the player using Bossa on the *Matrix Reloaded* trailer with and without reservations when competing with Linux kernel compilation. The behavior is represented as the difference in the percentage of the complete audio and video that has been treated so far. In both cases, we have given the X process a Linux real-time priority, so that when the player blocks to allow the video display, the X process runs immediately, thus reducing its impact as a performance bottleneck. Under the Linux 2.4 scheduling policy, the video falls far behind the audio in the presence of kernel compilation (third row of Table 1). With EDF, the player maintains correct synchronization.

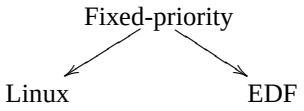


Figure 4. A scheduling hierarchy for use with MPEG video display

5 Conclusion

In this paper, we have given an overview of the Bossa framework to facilitate the development of kernel schedulers. Our approach is based on a the use of a DSL that simplifies programming and allows critical properties to be verified at compile time.

We have demonstrated the expressiveness of our approach by implementing several well-known scheduling policies in Bossa. Our initial experience with the Bossa compiler has shown that it is useful in catching both common inattention errors and errors related to incorrect understanding of the target OS. Since integration of a policy into the kernel is handled by the compiler and the framework, it is easy to test new policy variants. Thus, scheduler programming is made accessible to non-kernel experts. Indeed, we have developed lab materials for teaching basic scheduling principles to undergraduates using Bossa and have used these materials in graduate and undergraduate courses over the past few years. In our experience, the ease of use and robustness of the DSL has allowed students to freely experiment with scheduling at the OS kernel level without crashing the machine.

We are currently porting Bossa to the real-time OS Jaluna and are considering how to extend Bossa to multiprocessors. Finally, based on our success in using AOP to integrate the Bossa framework into an existing OS, we are currently studying how to use similar techniques to implement other kinds of OS evolutions [12].

Availability Bossa and all material described in this paper are available at the Bossa web site: `http://www.emn.fr/x-info/bossa/` .

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in

KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, Phoenix, AZ, Dec. 1999.

[3] J. L. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, pages 63–73, Seattle, WA, Nov. 1997.

[4] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.

[5] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.

[6] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[7] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, Dec. 1998.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997.

[9] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, 2001.

[10] J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, number 3286 in Lecture Notes in Computer Science, pages 436–455, Vancouver, Canada, Oct. 2004.

[11] J. L. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies (invited paper). In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04*, pages 80–91, Verona, Italy, Aug. 2004.

[12] J. L. Lawall, G. Muller, and R. Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, Mar. 2005.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[14] J. R. Lorch and A. J. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, Oct. 1997.

[15] G. Muller, J. Lawall, J.-M. Menaud, and M. Südholt. Constructing component-based extension interfaces in legacy systems code. In *ACM SIGOPS European Workshop 2004 (EW2004)*, pages 80–85, Leuven, Belgium, Sept. 2004.

[16] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005 - High Assurance Systems Engineering Conference*, Heidelberg, Germany, Oct. 2005. To appear.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

[18] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, Saint-Malo, France, Oct. 1997.

[19] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [20], pages 141–148.

[20] *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.

[21] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference (DAC'99)*, pages 134–139, New Orleans, LA, June 1999.

[22] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–158, New Orleans, LA, Feb. 1999.

[23] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4):475–488, Aug. 1997.

[24] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 149–163, Bolton Landing (Lake George), NY, Oct. 2003.

[25] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [20], pages 149–156.

Les langages de description d'architecture pour le temps réel

Anne-Marie Déplanche, Sébastien Faucou
Institut de Recherche en Communications et Cybernétique de Nantes (UMR n° 6597)
CNRS / École Centrale de Nantes, École des Mines de Nantes, Université de Nantes
Prenom.Nom@irccyn.ec-nantes.fr

Résumé

En accompagnement de la notion d'architecture logicielle en tant que « perspective haut niveau d'un système logiciel » et reconnue comme le cœur d'une discipline à part entière, des formalismes sont apparus au cours des années 90 : les ADLs (« Architecture Description Languages », soit langages de description d'architecture). Dotés d'une syntaxe et d'une sémantique bien définies, ils ont été proposés en remplacement des nombreuses notations informelles utilisées jusqu'alors pour décrire la structure des systèmes logiciels. De nombreux ADLs ayant été proposés par le milieu académique, l'article présente les concepts et abstractions caractéristiques que la plupart d'entre eux partagent. Il montre ensuite comment, dans le domaine des systèmes temps réel, une démarche architecturale et l'utilisation de langages adaptés tels que les ADLs, qui favorisent une approche globale, sont particulièrement intéressants. Cette argumentation s'appuie sur des ADLs spécialement conçus pour répondre aux besoins spécifiques du domaine temps réel.

1. Introduction

C'est dans les années 70, lorsque les systèmes logiciels ont commencé à franchir un certain seuil de complexité que l'idée de conception architecturale (ou encore « programming-in-the-large »), au sens d'une activité de conception séparée de la conception détaillée (ou encore « programming-in-the-small »), a émergé [8]. La notion d'architecture logicielle, en tant que « perspective haut niveau d'un système logiciel », n'apparaît réellement qu'à partir des années 90 et est alors présentée comme le cœur d'une discipline à part entière [22].

Même si tous s'accordent sur le fait qu'une architecture (logicielle) relève de la structure gros grain d'un système et de l'organisation de logiciels, il n'y a pas de consensus général sur une définition précise de ce terme. Ce terme désigne pour certains la description des structures et organisations des logiciels d'un système (ou d'une classe de système) particulier, tandis que pour d'autres, il désigne une discipline, un champ d'étude. À titre d'exemples, voici quelques définitions rencontrées dans la littérature concernée :

- *« ... it not only reflects how the functional requirements are met, but addresses non-functional requirements, design rationale, architecture style. »* [34]
- *« ... a view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission. »* [7]
- *« ... the structure or structures of the system, which compromise software components, the external visible properties of those components, and the relationships among them. »* [4]
- *« ... the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. »* [25]
- *« While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high level analysis and critical appraisal. »* [20]

Enfin, dans [35], Perry et Wolf souhaitent mettre en place les fondations de la recherche future sur les architectures logicielles et proposent pour cela un modèle (accompagné d'une caractérisation des architectures et styles architecturaux logiciels) qu'ils expriment sous forme condensée par le triplet : *Software architecture* = { *Elements, Form, Rationale* } (soit encore, une architecture logicielle est un ensemble d'éléments architecturaux accompagnés de propriétés et relations qui en contraignent le choix et l'organisation, sans oublier les motivations expliquées de l'architecte), plaçant ainsi en quelque sorte l'architecture

logicielle à mi-chemin entre les exigences et la conception.

Les enjeux classiquement mis en avant pour motiver l'émergence de l'approche architecturale sont :

- l'architecture constitue un cadre pour maîtriser la complexité d'un système, dans le but de répondre à toutes les exigences du cahier des charges, qu'elles soient fonctionnelles ou non ;
- l'architecture est une base pour la conception comme pour l'estimation des coûts et la gestion de projet ;
- l'architecture offre un support pour la prise en considération de l'évolutivité et de la réutilisation des systèmes logiciels ;
- l'architecture est une base de raisonnement à des fins d'analyse et de validation.

Ces arguments font apparaître clairement le rôle joué par la description de l'architecture au sein du processus de développement d'un système logiciel : au-delà du résultat de l'étape de conception générale, elle a vocation à servir de référentiel aux étapes suivantes, qui viennent y puiser des informations (extraction d'un modèle formel à des fins d'analyse, extraction de spécifications détaillées à des fins de codage, etc.) et l'enrichir de nouvelles informations en retour. Ainsi, en fonction du stade de développement du système, la description de l'architecture peut être une spécification abstraite du système qu'on cherche à concevoir ou bien une description détaillée du système opérationnel.

Pour permettre la mise en œuvre de cette approche, différentes notations ont été proposées. Trois générations sont identifiées dans la littérature :

- les *langages d'interconnexion de modules* ou MIL (« Module Interconnection Language ») tout d'abord : ils permettent de décrire (au travers des services fournis et requis) les modules logiciels composant une application implémentée, les dépendances entre (les services de) ces modules comme leur implantation physique sur les machines. Ils sont très souvent associés à des bus logiciels assurant l'installation des modules, leur interconnexion comme la gestion de leurs échanges. On trouve dans [36] un panorama de ces langages ;
- les *langages de configuration* ensuite : les entités logicielles ici manipulées sont plus « abstraites » ; on parle généralement de (types de) composants présentant une interface bien définie composée de points d'interaction (au sens large), instanciables, pouvant être primitifs ou composites selon une structuration hiérarchique. La composition est un mécanisme fondamental introduit par ces langages ; elle repose sur l'interconnexion de composants via leurs interfaces. Restant toutefois fortement lié à l'implémentation, le langage de configuration est généralement associé à un langage de programmation détaillée pour le codage des composants primitifs. L'objectif majeur alors visé est celui de la configurabilité, c'est-à-dire la capacité de spécifier et de modifier la configuration d'un système (par ajout, suppression ou remplacement de composants) de manière statique (hors ligne) ou dynamique (en ligne). Dans ce dernier cas, le langage de configuration permet de décrire les changements à opérer et les conditions sous lesquelles ils peuvent avoir lieu. Parmi les langages de configuration les plus connus (en leur temps), on peut citer Conic [26], à l'origine de Darwin [28, 29], ou encore Durra [3] ;
- petit-à-petit, les langages de configuration ont pris une orientation plus conceptuelle et plus formelle, tant pour les composants que pour les connexions entre composants, ceci afin d'accroître les possibilités de raisonnement (évaluation et/ou analyse) au niveau architectural. Une sémantique formelle s'est substituée aux simples conventions de représentation. Est alors apparue, au cours des années 90, la dénomination de *langage de description d'architecture* ou ADL (« Architecture Description Language »). De nombreux ADLs ont été développés par le milieu académique à cette époque : Wright, Darwin, Unicon, Rapide, Aesop, C2 SADL, MetaH, etc. [32]. Il s'est agi là de contributions bien souvent complémentaires mais isolées, rendant difficiles la mise en commun de descriptions architecturales comme l'intégration des outils associés. Le langage ACME [21] est une tentative de réponse, mais dans la pratique il n'offre un support satisfaisant que pour l'échange des aspects structurels des descriptions. On peut cependant penser que l'essor actuel des technologies de transformation de modèle va permettre d'offrir à court terme des techniques adaptées à la résolution de ce problème.

Le domaine des systèmes temps réel affiche des besoins qui justifient actuellement une réelle réflexion sur l'approche de conception architecturale : organisation complexe (présence de fonctionnalités multiples interdépendantes), architectures matérielles réparties, présence de contraintes non-fonctionnelles qui lient intimement éléments logiciels et matériels, utilisation optimisée des ressources, reconfigurabilité dynamique, prédictibilité et donc nécessité de vérification a priori et au plus vite dans le processus de développement, etc. De plus, la généralisation de l'utilisation des systèmes temps réel embarqués dans des domaines comme l'automobile ou l'avionique (où les produits sont déclinés en gamme et sont construits par assemblage de sous-systèmes fournis par différents équipementiers) fait émerger des exigences nouvelles

de flexibilité, réutilisabilité, portabilité, interopérabilité, etc. Pour être respectées, de telles exigences doivent être prises en considération dès la conception de haut niveau. Des travaux de recherche ont été et sont encore menés autour d'un processus de développement « basé architecture » pour la conception de systèmes temps réel et quelques ADLs dédiés sont apparus comme MetaH [5, 39], CLARA [12, 17, 18] , COTRE [16] ou encore EAST-ADL [9]. Récemment, le SAE[1] a normalisé un ADL « temps réel » destiné en premieur lieu au domaine de l'avionique : AADL[2] [37] (« Architecture Analysis and Design Language »). Enfin, ces thèmes ont été au cœur des travaux de l'Action Spécifique CAT (pour Composants et Architectures Temps réel) dont les travaux sont résumés dans [13].

La présentation qui suit est constituée de deux grandes parties. Pour commencer (section 2), nous mettons en place les concepts fondamentaux qui définissent un ADL généraliste et nous les illustrons par un exemple traité avec le langage Wright. Dans la suite, nous ciblons notre propos sur le domaine du temps réel. En section 3 quelques-uns des aspects fondamentaux des systèmes temps réel sont rappelés, qui ne peuvent être ignorés lors de la conception architecturale et qui doivent donc être supportés par un ADL « temps réel ». La complexité résultant de la prise en considération de ces spécificités amène naturellement à la notion de vue architecturale qui est brièvement discutée. Suivent les présentations par le traitement d'un même exemple de deux ADLs temps réel : CLARA (section 4) et MetaH (section 5). En complément, un rapide aperçu de la proposition COTRE (section 6), avant les conclusions (section 7).

2. Les langages de description d'architecture

2.1. Les concepts à la base des ADLs

Les ADLs constituent une classe de langages offrant des abstractions pour la description « gros grain » des systèmes logiciels. Comme indiqué en introduction, de multiples langages appartiennent à cette catégorie, langages qui, pour certains, diffèrent de manière majeure ne serait-ce que par leur syntaxe, leur sémantique, leur expressivité et les buts qu'ils visent. En proposer une définition unique, précise et consensuelle est alors un problème délicat. Pour y répondre, dans un article de 2000 qui fait référence dans le domaine [32], Medvidovic et Taylor ont préféré établir un cadre de classification et de comparaison des ADLs. Dans ce contexte, ils ont proposé un « test » permettant de savoir si un langage est un ADL, fondé sur la présence dans le langage de quatre concepts : **composant**, **interface**, **connecteur** et **configuration**. Nous résumons ci-après les définitions qu'ils donnent de ces concepts.

Un **composant** est une unité de traitement ou de stockage de donnée. C'est une entité dynamique dont l'état évolue au cours du temps, en fonction des sollicitations qu'elle reçoit. Avec les connecteurs, les composants constituent les briques de base utilisées pour décrire le système. Ainsi, pour faciliter la réutilisation d'un composant au sein d'une même architecture ou d'architectures différentes, les ADLs font généralement la différence entre type et instance de composant. Par ailleurs, l'accent est porté sur les possibilités d'utilisation du composants par des tiers, et non sur son fonctionnement interne. Cela explique l'importance donnée à la notion d'interface. Enfin, même si les ADLs ont comme but premier de décrire la structure d'un système, une sémantique est souvent associée au composant, qui décrit son comportement (i.e. les traitements réalisés et événements produits en réaction aux sollicitations reçues à l'interface). Lorsqu'il s'agit d'un composant atomique, le comportement est spécifié en langage naturel, algorithmique ou formel (à des fins de vérification). Pour les composants non atomiques, il est défini par une architecture (configuration) encapsulée.

L'**interface** (ou les interfaces) regroupe l'ensemble des points d'interaction (au sens large) du composant avec son environnement. Pour refléter la nature de l'interaction, ces points sont « orientés » : port en entrée ou en sortie, service offert ou requis, etc. Par ailleurs, un ensemble de contraintes et de propriétés sont associées au composant, qui définissent ses conditions d'utilisation (notion de contrat).

Un **connecteur** est assimilable à un composant spécialisé dans les interactions : il permet de mettre en relation (via leurs interfaces) un ensemble de composants et spécifie les règles qui régissent cette interaction. Ici aussi on distingue type et instance de connecteur. Un connecteur possède une interface constituée de points d'interaction appelés rôles (en référence au rôle joué par le composant qui y est attaché dans l'interaction). Ils sont destinés à être connectés aux points d'interaction des composants (sous réserve de compatibilité entre le point d'interaction et le rôle). La liaison entre les différents rôles est assurée par la "glu" : il s'agit d'une spécification du comportement du connecteur, similaire à la description sémantique d'un composant. Le fait de considérer les connecteurs comme des entités de première classe est une avancée issue du monde des ADLs. On retrouve aujourd'hui cette notion de connecteur dans UML2.

Une **configuration** est un graphe bipartite (instances de) composants - (instances de) connecteurs qui spécifie tout ou partie de l'architecture du système selon un certain point de vue. On peut ainsi avoir une

[1] SAE : Society of Automotive Engineers, `http://www.sae.org`.

[2] Ce langage (promu en partie par l'industrie) faisant l'objet d'un autre chapitre de ce recueil, il n'est pas détaillé ici.

configuration qui exprime la connectivité entre les différentes briques logicielles, une configuration qui exprime les flots de contrôle concurrents qui traversent ces briques et leur répartition sur les ressources d'exécution, etc. La construction d'une configuration est régie par des règles qui garantissent que le système décrit vérifie un certain nombre de bonnes propriétés : typage au niveau des flots de données, absence de deadlocks, respect d'une heuristique de conception (on parle alors de *style* d'architecture, par exemple le style « pipe and filter »). Certaines de ces propriétés sont inhérentes au langage (typage des flots de données), d'autres doivent être exprimées sous forme de contraintes spécifiées au niveau de la configuration (adhésion à un style) et / ou vérifiées a posteriori (absence de deadlock). Les configurations expriment la composition des briques logicielles, celle-ci pouvant être horizontale (interconnexion) ou verticale (abstraction / encapsulation).

2.2. Un exemple

En prenant comme support l'ADL WRIGHT [2], nous allons illustrer les notions introduites ci-dessus.

WRIGHT est un ADL défini par R. Allen et D. Garlan, en 1997, à l'Université de Carnegie-Mellon [2]. Il est destiné à des applications logicielles « classiques » et est donc présenté comme un ADL généraliste. Il offre uniquement une syntaxe textuelle et fournit une base formelle pour la description de configurations architecturales. Son utilisation ici se justifie principalement par la simplicité de ses concepts structurants. Ses caractéristiques majeures peuvent se résumer ainsi :

- les connecteurs font l'objet de spécifications explicites et indépendantes ;
- le comportement abstrait des composants et connecteurs peut être décrit à l'aide d'un sous-ensemble de l'algèbre de processus CSP[3] [24] (nous supposons, pour la suite, que le lecteur possède une connaissance minimum de CSP) ;
- de par la sémantique formelle de cet ADL, la spécification d'une architecture peut donner lieu à un certain nombre de vérifications statiques. Ainsi, une description WRIGHT peut être traduite en CSP puis analysée avec des outils comme FDR [19] pour vérifier la compatibilité d'une connexion (respect d'un même protocole par un composant et un connecteur auquel il est lié) ou encore l'absence de deadlock.

Nous traitons ci-après en WRIGHT l'exemple bien connu du dîner des philosophes. Notre présentation s'inspire de celle donnée dans [30]. La simplicité du problème ne permet pas de présenter l'intégralité des possibilités du langage, le lecteur intéressé se reportera à [2] pour plus de détails et compléments.

Le système à décrire est le suivant : des philosophes se réunissent pour philosopher et dîner. Le dîner est constitué d'un plat de spaghetti qui, selon les coutumes de ces philosophes, se mange à l'aide de deux fourchettes pour un convive. Une table est dressée, qui comporte une assiette par philosophe et une fourchette par assiette. Chaque philosophe pense puis mange puis pense, etc. Il saisit les fourchettes une par une ; il doit posséder les deux fourchettes qui entourent son assiette pour pouvoir manger. Le problème est habituellement de définir un protocole de synchronisation des actions des philosophes de façon à éviter aux philosophes la famine. Étant donné l'objectif de notre exposé, on se concentre ici sur la description de la « structure » du système et la description présentée comporte un deadlock.

Les composants sont les philosophes et les fourchettes. Les connecteurs sont les mains des philosophes destinées à associer mangeurs et outils. Chaque philosophe possède deux bras : son interface est donc composée de deux ports gauche et droite. Chaque fourchette possède un manche : son interface est donc composée d'un port manche. Une main relie un mangeur à un outil : son interface fait état de deux rôles mangeur et outil. Lorsqu'on met en place une table de philosophes, il s'agit de définir le schéma d'interaction entre les philosophes et les fourchettes, soit encore d'associer chaque bras de chaque philosophe à une fourchette par l'intermédiaire d'une main. Pour ce faire, chaque port d'un philosophe est connecté à une main avec un rôle mangeur, tandis qu'une fourchette est connectée à la même main avec un rôle outil.

Interface Type $Bras = \overline{prendre} \rightarrow \overline{deposer} \rightarrow Bras \sqcap \S$
Component *Philo*
  Port $Gauche = Bras$
  Port $Droit = Bras$
  Computation $= penser \rightarrow \overline{Gauche.prendre} \rightarrow$
    $\overline{Droit.prendre} \rightarrow manger \rightarrow \overline{Gauche.deposer} \rightarrow$
    $\overline{Droit.deposer} \rightarrow$ **Computation** $\sqcap \S$

FIG. **1. Le composant** *Philo*

Le code WRIGHT qui décrit le composant *Philo* est donné fig. 1. Ce composant a deux ports similaires (les « bras » du philosophe), on définit donc un **Interface Type** pour capturer le comportement commun (sous la forme d'un processus CSP). Outre la définition de son interface, la spécification du comportement du composant est donnée à travers la clause **Computation**.

La description du composant *Fourchette* est donnée par la fig. 2 selon le même principe.

Le code WRIGHT du connecteur *Main* est donné fig. 3. Ce connecteur possède deux rôles dont on spé-

[3] Par rapport à CSP « classique », WRIGHT fait la différence entre les communications initiées par un processus (surlignées) et celles initiées par son environnement (non surlignées). Par ailleurs, § est utilisé comme abbréviation de $\checkmark \rightarrow Stop$ pour marquer la terminaison correcte d'un processus.

Component *Fourchette*
  Port $Manche = pris \rightarrow depose \rightarrow Manche \sqcap \S$
  Computation $= Manche.pris \rightarrow Manche.depose \rightarrow$
    Computation $\sqcap \S$

FIG. **2. Le composant** *Fourchette*

Connector *Main*
  Role $Mangeur = \overline{prendre} \rightarrow \overline{deposer} \rightarrow Mangeur \sqcap \S$
  Role $Outil = pris \rightarrow depose \rightarrow Outil \sqcap \S$
  Glue $= Mangeur.prendre \rightarrow \overline{Outil.pris} \rightarrow$ **Glue**
    $\sqcap\ Mangeur.deposer \rightarrow \overline{Outil.depose} \rightarrow$ **Glue**
    $\sqcap \S$

FIG. **3. Le connecteur** *Main*

cifie le comportement sous la forme de deux processus CSP. Le lien entre ces deux processus est assuré par le processus spécifié dans la clause **Glue** : lorsque le rôle *Mangeur* reçoit le message *prendre*, le message *pris* est émis vers le rôle *Outil*.

Configuration *Diner*
  Component *Philo* ...
  Component *Fourchette* ...
  Connector *Main* ...
  Instances
    $p1, p2, p3 :$ *Philo*
    $f1, f2, f3 :$ *Fourchette*
    $m11, m12, m21, m22, m31, m32 :$ *Main*
  Attachements
    $p1.Gauche$ **as** $m11.Mangeur$
    $f1.Manche$ **as** $m11.Outil$
    $p1.Droit$ **as** $m12.Mangeur$
    $f2.Manche$ **as** $m12.Outil$
    $p2.Gauche$ **as** $m21.Mangeur$
    $f2.Manche$ **as** $m21.Outil$
    $p2.Droit$ **as** $m22.Mangeur$
    ...
End Diner

FIG. **4. Un extrait de configuration (3 philosophes)**

Le système est décrit au sein de la configuration *Diner* (fig. 4). Après la définition des types de composant et de connecteur (telle que détaillée ci-dessus et non rappelée ici), la rubrique **Instances** permet de spécifier les instances utilisées pour décrire le système. Ces instances sont ensuite connectées dans la rubrique **Attachements** (un port d'une instance de composant à un rôle d'une instance de connecteur).

L'exemple ci-dessus ne nous permet pas d'illustrer toute les possibilités offertes par WRIGHT comme la composition hiérarchique (la clause **Computation** d'un composant est alors définie par une configuration) ou encore la possibilité de définir formellement des styles d'architecture. Dans WRIGHT, un style regroupe un vocabulaire (une ontologie) et un ensemble de contraintes syntaxiques (pour contraindre la topologie d'une architecture relevant de ce style) et sémantique (pour contraindre le comportement abstrait des composants ou connecteurs appartenant à ce style), les contraintes étant exprimées à l'aide de formules de la logique des prédicats d'ordre 1 et de la théorie des ensembles appliquée à des ensembles de base se rapportant aux entités constituant une spécification WRIGHT (par exemple : Connector est l'ensemble des connecteurs d'une configuration, $Traces(P)$ est l'ensemble des traces du processus P, etc.).

D'autre part, dans sa version originale, WRIGHT ne supportait pas la spécification des aspects dynamiques d'une architecture (instanciation /suppression de composant par exemple). Il a été étendu depuis [1], comme *C2* [33] ou encore Darwin [28] qui apportent des réponses plus ou moins complètes à de tels besoins. Pour cela, des événements particuliers « de contrôle » sont introduits et peuvent être référencés dans la description d'un port, permettant ainsi de décrire dans quelles conditions un composant accepte et traite les reconfigurations. Ces événements de contrôle sont utilisés par ailleurs dans une vue séparée, un programme de configuration ou configuror, décrivant comment ces événements déclenchent les reconfigurations. Là encore, en raison de la sémantique formelle associée, des analyses de cohérence peuvent être engagées.

3. ADLs et systèmes temps réel

Comme indiqué en introduction, les systèmes temps réel modernes affichent une complexité croissante du point de vue fonctionnel mais aussi extra-fonctionnel (architecture matérielle, support d'exécution, exigences temporelles, de sûreté de fonctionnement, de consommation, interdépendance de ces différentes composantes, etc.). Ceci justifie d'autant l'importance de la conception architecturale pour leur développement et de langages adaptés tels que les ADLs qui favorisent une approche globale et abstraite indispensable. Toutefois, de par leurs spécificités, les concepts de base des ADLs doivent être spécialisés et/ou enrichis afin d'offrir la possibilité d'inclure des informations pertinentes et indispensables au processus de développement consécutif. Nous nous proposons ci-après de mettre en avant quelques-uns des aspects qui nous paraissent fondamentaux. Par les exemples d'ADLs plus particulièrement dédiés au temps réel qui font suite, certains d'entre eux sont illustrés.

Les systèmes temps réel présentent de manière générale un **caractère réactif** vis-à-vis d'événements internes (signal de synchronisation produit par un composant du système, terminaison de l'exécution d'un composant, occurrence d'une situation d'exception, etc.), ou issus de l'environnement contrôlé, ou encore liés au temps (signal d'horloge périodique, expiration d'un chien de garde, etc.). Si l'on souhaite être à même de réaliser des analyses comportemen-

tales, il est primordial de pouvoir décrire, dès le niveau « architecture fonctionnelle », les liens entre les flots de contrôle qui parcourent le système et ces différents événements.

La description de l'**environnement physique** contrôlé ou du moins de ses interactions visibles avec le système relève également des informations à inclure dans la description architecturale d'un système temps réel. D'une part dans le but de pouvoir valider l'architecture proposée par rapport à son futur environnement opérationnel. D'autre part, l'interface de l'environnement n'est pas toujours fixée préalablement à la conception ; il existe alors différentes possibilités d'instrumentation du procédé dont le choix influence la structure et le comportement du système.

Le **temps** est une dimension fondamentale de ces systèmes. Sa manipulation est indispensable dès le niveau architectural, soit qu'il participe au flot de contrôle, soit qu'il quantifie des propriétés temporelles de composants ou de configurations, soit des contraintes temporelles. Issues du cahier des charges, elles sont progressivement dérivées et affinées au cours du développement du système. Les identifier au niveau de la description architecturale assure la documentation et la traçabilité entre les différentes étapes de développement, mais surtout guide les décisions de mise en œuvre (partitionnement, placement, ordonnancement, etc.).

L'**architecture matérielle** comme le **support d'exécution** occupent une place importante dans la conception d'un système temps réel. Leur spécification ne peut donc pas être exclue de la description architecturale et du processus de développement consécutif. D'une part, dans le domaine des applications visées, la définition comme le dimensionnement de ces éléments ne sont pas « libres » (contraintes technologiques diverses, minimisation des coûts, interopérabilité avec des systèmes existants, ressources disponibles limitées, etc.) et constituent déjà en soi un réel problème dès l'étape de conception architecturale. D'autre part, un couplage très fort existe entre logiciels applicatifs et plate-forme support (au sens large). Le comportement du système temps réel, tant qualitativement que quantitativement, dépend bien sûr des services des exécutifs, des middlewares, des protocoles de communication, etc. utilisés, mais aussi de sa nature distribuée ou non, de la capacité de traitement des unités de calcul, de la bande passante des canaux de transmission, des modes de gestion de la mémoire, etc. À moins de procéder par approximations (grossières et parfois insensées), il est donc impossible d'envisager une analyse de performances temporelles comme de sûreté de fonctionnement sur l'architecture du système en construction sans traiter conjointement aspects logiciels et matériels (y compris l'environnement). Sans aller jusqu'à une description aussi riche et fine que celle faite par ailleurs dans le domaine de la conception de circuits (serait-elle exploitable au niveau de la conception architecturale telle que vue ici ?), il apparaît indispensable d'inclure explicitement cette composante dans la description architecturale d'un système temps réel.

Il est fréquent pour un système (de contrôle) temps réel d'adopter des configurations de fonctionnement différentes pendant sa vie opérationnelle. Ces configurations peuvent être liées à des phases différentes de son exploitation ; un système avionique par exemple doit offrir des fonctionnalités spécifiques selon que l'avion est en phase de décollage, de vol ou d'atterrissage. Il peut également s'agir d'évolutions liées à la détection et au traitement de situations d'exception liées à l'occurrence de fautes logicielles ou matérielles du système de contrôle, voire de défaillances d'une partie du procédé ; il faut dans ce cas assurer un service dégradé garantissant l'intégrité du contrôle. Ces différentes configurations sont regroupées dans des **modes de fonctionnement**. Très souvent, le service rendu par un système temps réel se décline ainsi selon différents modes de fonctionnement à activer ou désactiver selon les besoins. La spécification de tels modes et des stratégies appropriées de commutation relève (en partie) du niveau architectural. Il est donc souhaitable qu'un ADL pour le temps réel offre nativement des abstractions adaptées à ces besoins.

Enfin, est-il nécessaire de rappeler que la conception des systèmes temps réel exige non seulement des langages présentant une expressivité adéquate mais aussi des **outils** associés ? Ainsi la possibilité de mener des vérifications de propriétés comportementales (temporelles comme de sûreté de fonctionnement au sens large) au niveau architectural apparaît particulièrement intéressante puisqu'elle profite d'une vision globale du système et doit donc permettre de détecter des erreurs de conception très tôt dans le processus de développement, minimisant ainsi le coût de leur correction. Par ailleurs, la description architecturale d'un système donnée par un ADL est le point d'entrée de l'étape de déploiement. Selon son niveau d'abstraction et d'indépendance vis-à-vis de la plateforme cible, le déploiement d'une description architecturale peut aller de la simple traduction à un travail complexe de construction. L'architecte doit alors être assisté par des outils d'exploration de l'espace des implémentations et de génération/configuration de code. Il apparaît finalement que l'utilisation d'un ADL permet à la fois de dériver d'une même racine commune différents modèles tant pour la vérification/validation que pour l'implantation, et peut donc être un moyen pour de la cassure sémantique ou pour le moins de faciliter la traçabilité entre ces niveaux.

Nous venons d'énumérer un certain nombre d'aspects des systèmes temps réel dont l'expression au niveau architectural est nécessaire. Il est bien entendu qu'une description regroupant simultanément

l'ensemble de ces informations est difficilement réalisable et compréhensible. Il est alors convenu de décomposer la description selon différentes *vues architecturales*, c'est-à-dire selon différentes perspectives pour lesquelles on élimine les entités non concernées au profit de celles directement impliquées dans le point de vue considéré. Le but recherché est de faciliter la démarche de conception en traitant séparément les différents problémes sous-jacents. S'il n'existe pas de réel standard en matière de vue architecturale, le modèle le plus largement accepté par la communauté du génie logiciel (car étroitement lié à UML) est le modèle *« 4+1 views »* (avec ses variantes) [27] qui distingue 5 vues : *Logical view*, *Implementation view*, *Process view*, *Deployment view* et *Use-case view*.

Dans sa thèse [41], A. Wall souligne qu'il est difficile, voire impossible, de décomposer toutes les caractéristiques d'un système en vues parfaitement distinctes. Il préfère donc décliner pour ces différentes vues, trois *« aspects »* particulièrement sensibles pour le domaine des systèmes temps réel (un même aspect étant éventuellement représenté dans plusieurs vues) : *« aspect temporel »*, *« aspect communication »* et *« aspect synchronisation »*.

Dans le même ordre d'idée, dans [17], S. Faucou reprend les trois composantes majeures impliquées dans la conception architecturale d'un système temps réel, à savoir l'*architecture logicielle*, l'*architecture support* et l'*architecture opérationnelle*, et recense pour chacune d'entre elles un certain nombre de vues pertinentes :

- pour l'architecture logicielle : la vue *« composant »* s'attache à la définition individuelle des composants de l'application, tant au niveau de leurs interfaces que de leurs comportements internes ; la vue *« structurelle »* montre l'organisation générale du système, par l'interconnexion des composants et des connecteurs qui expriment la nature des interactions ; la vue *« réactive »* regroupe la description des réactions face aux occurrences d'événements visibles au niveau architectural, établissant ainsi les flots de contrôle des traitements réalisés par le système ;
- pour l'architecture support : la vue *« support logiciel d'exécution »* décrit les services système offerts aux logiciels applicatifs comme les services des exécutifs, d'un middleware, les services de niveau application des protocoles de communication, etc. ; la vue *« support physique d'exécution »* définit et caractérise la constitution physique du système en termes de ressources matérielles et de topologie ;
- pour l'architecture opérationnelle : la vue *« implémentation »* spécifie la projection de l'architecture logicielle sur le support logiciel d'exécution, c'est-à-dire l'implantation des objets et mécanismes de haut niveau de la première sur les objets et services concrets proposés par le second ; la vue *« système »* définit la projection des éléments de la vue d'implémentation sur ceux de la vue support physique d'exécution en termes de placement des éléments logiciels (tâches, variables, messages), d'ordonnancement et optimisation des accès aux ressources matérielles (processeurs et réseaux), de configuration du support logiciel d'exécution, etc. C'est uniquement à partir d'une telle vue (qui est la première à intégrer tous les aspects logiciels, matériels et environnementaux) que l'étude du comportement temporel du système peut être réellement envisagée.

Un point de vue similaire a été adopté par le projet EAST-EEA[4] dont le contexte est celui des systèmes embarqués dans les véhicules terrestres.L'un des objectifs de ce projet a été de définir un langage de description des architectures, EAST-ADL, pour pouvoir décrire de manière plus efficace, et couplée au processus de développement, l'informatique embarquée dans les véhicules. Cinq niveaux d'abstraction ont été proposés et permettent de décrire les fonctionnalités du véhicule et leur organisation depuis un point de vue élevé (ce que voit l'utilisateur : les prestations du véhicule) jusqu'à un niveau très bas (celui des tâches et des messages). On y trouve différentes vues énumérées ci-après (les quatre premières relèvent du logiciel d'application tandis que les trois dernières sont dédiées à l'implémentation) : *« vehicule view »* , *« functional analysis architecture »* , *« functional design architecture »* , *« logical architecture »* , *« hardware architecture »* , *« technical architecture »* et *« operational architecture »* .

Dans les deux sections qui suivent, nous présentons deux ADLs temps réel : CLARA et MetaH. Ces présentations reposent sur un exemple (fictif et simple) d'application de pilotage numérique décrit dans le paragraphe suivant. Pour chaque langage, nous évaluons la façon dont il répond à la liste d'exigence établie ci-dessus.

Le système à spécifier possède deux entrées (une mesure réalisée sur le procédé piloté et une consigne transmise par l'opérateur fixant l'état à atteindre) et une sortie (une action à réaliser sur le procédé piloté). C'est un système échantillonné. Fonctionnellement, le système de pilotage doit : traduire le format brut délivré par le capteur en un format adapter aux calculs les calculs, puis calculer la commande (fonction de la mesure, de la valeur courante de la consigne et de l'état estimé du procédé), estimer le nouvel état et enfin traduire la commande en un ordre compréhensible par l'actionneur. La période d'échantillon-

[4]EAST-EEA : Electronic Architecture and Software Technology - Embedded Electronic Architecture. Projet ITEA de juin 2001 à juin 2004, impliquant un consortium de constructeurs automobiles, d'équipementiers automobiles et d'universitaires : `http://www.east-eea.net`.

nage est fixée à 500ms avec une gigue acceptable de 20ms. On souhaite par ailleurs que l'action soit notifiée à l'actionneur au plus tard 250ms après l'instant d'échantillonnage.

La figure 5 est la description d'une architecture logicielle pour ce système, décrite à l'aide la syntaxe graphique de CLARA. Cette syntaxe intuitive nous permet de présenter notre architecture candidate avant d'entrer dans les détails des langages. Nous avons choisi de découper le système en quatre composants, correspondant aux quatre activités suivantes : traduction de la mesure, calcul de la commande, calcul du nouvel état et traduction de la commande. Le système est périodique et les synchronisations entre les composants sont ensuite établies par le flot de données. Ce flot n'établit pas d'ordre pour l'exécution des deux dernières activités (calcul du nouvel état et traduction de la commande), ce choix étant laissé aux étapes ultérieures de la conception (choix du nombre des unités d'ordonnancement, de la politique sélectionnée et des valeurs des paramètres associés).

4. CLARA

La spécification complète de l'exemple traité est donnée en annexe A.1.

4.1. Présentation

CLARA [12, 18] (Configuration LAnguage for Real-time Application) est un ADL créé dans les années 90 au sein de l'équipe « Systèmes Temps Réel » de l'IRCCyN. Comme le montrent les figures 5 et 8 qui décrivent (quasiment) la même chose, il possède une syntaxe graphique et une syntaxe textuelle. Il permet la description à un haut niveau d'abstraction de l'architecture fonctionnelle d'applications temps réel. L'accent est particulièrement porté sur la spécification des synchronisations induites par les flots de contrôle et de données qui traversent les activités constitutives de l'application.

Les composants principaux sont les **activités**, qui peuvent être atomiques (associées à un traitement séquentiel fini) ou composées (englobent une configuration). Chaque activité possède deux interfaces : une interface implicite de contrôle constituée d'un port `START` et d'un port `END`, utilisée pour synchroniser le lancement (resp. se synchroniser sur la terminaison) du traitement associé à l'activité ; une interface explicite de transfert, composée de ports (orientés), utilisée pour permettre aux flots de données et de contrôle de traverser l'activité. Cette interface est spécifiée lors de la définition du type d'activité (fig. 6), de même que le comportement interne (plus d'explication sur ce sujet sont données en 4.2).

Au rang des autres composants figurent les variables et ressources partagées, ainsi que les **générateurs d'occurrences**. Ces derniers sont utilisés

```
TYPE_ACTIVITY T_CalculCommande;
    VAR_IN etat, consigne : t_etat;
           mesure : t_cmd_in;
    VAR_OUT commande : t_cmd_out;
    BEHAVIOR receive(mesure) -> receive(etat)
        -> receive(consigne)
        -> call(calculCmd, 10ms, 20ms)
        -> send(commande);
END_ACTIVITY;
```

FIG. **6. Définition d'un type d'activité**

comme source des flots de contrôle et de données. Ils produisent des occurrences selon une certaine loi et peuvent être internes au système (cas d'une horloge par exemple) ou externes (cas d'un périphérique attaché à un canal d'interruption par exemple). La figure 7 montre la définition d'un type de générateur périodique (utilisé ici pour cadencer le système).

```
TYPE_GENERATOR T_Horloge_500ms;
    PERIODIC 500ms;
    SIGNAL_OUT out;
    BEHAVIOR CLOCK;
END_GENERATOR;
```

FIG. **7. Définition d'un type de générateur d'occurrences**

Au sein des configurations, les (instances des) activités sont reliées par des **liens** construits autour de **protocoles**. Un protocole correspond à la notion de « glue » vue précédemment : il spécifie la politique de synchronisation entre l'entité jouant le rôle de producteur et l'entité jouant le rôle de consommateur dans cette interaction. Les **connecteurs** permettent de connecter un port d'une activité à un protocole (s'il s'agit d'un port en sortie, l'activité jouera le rôle producteur, s'il s'agit d'un port en entrée, l'activité jouera le rôle consommateur). Un connecteur peut être simple (association 1-1) ou complexe (association n-m, de diffusion ou de sélection). Dans une configuration, les liens sont spécifiés dans la rubrique `LINKS`. Ainsi, fig. 8, on a spécifié que :

- de `capture.mesure_out` vers `calculCommande.mesure_in` est établi un lien de transfert de données du type rendez-vous ;
- une donnée publiée par `calculCommande.commande` est : transmise vers `action.commande` via un rendez-vous ; transmise vers `majEtat.commande` via une boîte aux lettres ;
- de `H1.out` vers `capture.START` est établi un lien de transfert de signaux du type mémorisé (i.e. production asynchrone, consommation synchrone) ;
- etc.

La figure 8 correspond à la spécification du composant **système**, qui constitue la racine d'une des-

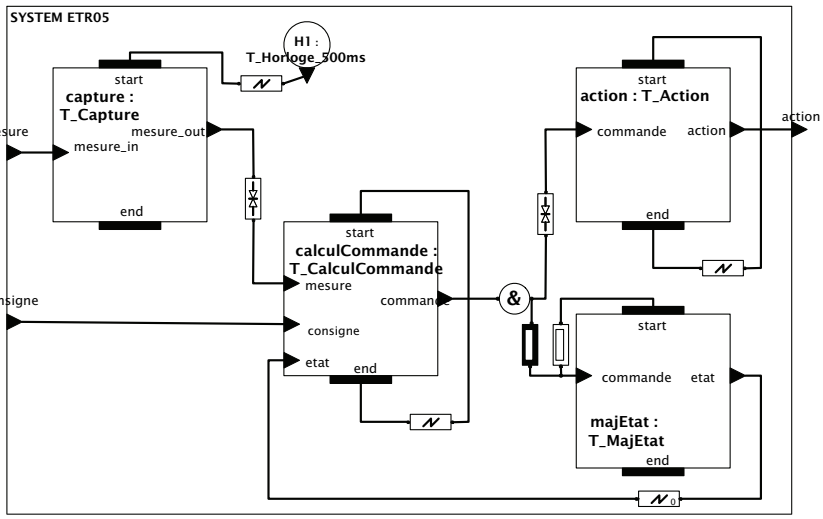


FIG. **5. Description de l'architecture fonctionnelle de l'exemple en CLARA (syntaxe graphique)**

```
SYSTEM Etr05;
VAR_IN mesure : t_mesure; consigne : t_etat;
VAR_OUT action : t_action;
GENERATOR H1 : T_Horloge_500ms;
ACTIVITY
    capture : T_Capture;
    calculCommande : T_CalculCommande;
    majEtat : T_MajEtat;
    action : T_Action;
LINKS
    // liens d'interface
    mesure TO capture.mesure_in;
    consigne TO calculCommande.consigne;
    action.action TO action;
    // liens inter-activites
    capture.mesure_out TO
        calculCommande.mesure : RDVd;
    calculCommande.commande TO
        (action.commande : RDVd
        & majEtat.commande : BAL);
    majEtat.etat TO calculCommande.etat : RAF0;
    // liens d'activation
    H1.out TO capture.start : MEM;
    calculCommande.end TO calculCommande.start : MEM;
    action.end TO action.start : END;
    majEtat.commande TO majEtat.start : INC;
CONSTRAINTS
    Abs(capture.mesure_in.cnf) < 25ms;
    480ms < capture.mesure_in.cnf < 520ms;
    0ms < capture.mesure_in.cnf :
          action.action.cnf     < 250ms;
END_SYSTEM.
```

FIG. **8. Définition de la configuration**

cription CLARA. Comme tout composant, il possède une interface. Son comportement, il est donné sous la forme d'une configuration : instanciation de composants encapsulés (rubriques `GENERATOR` et `ACTIVITY`), spécification des liens entre eux (rubrique `LINKS`) et finalement expression de contraintes (rubrique `CONSTRAINTS`).

4.2. Aspects réactifs

Dans CLARA, les aspects réactifs occupent une place prépondérante. Les flots de contrôle et de données du système sont ainsi entièrement modélisé. On spécifie comment ils passent d'un composant à un autre à travers les liens, mais également comment ils traversent les composants : lors de la définition d'un type d'activité atomique (cf. fig. 6), dans la rubrique `BEHAVIOR`, on spécifie le comportement sous la forme d'une séquence alternant interaction avec les flots de contrôle et de donnée (primitives `send`, `receive`, etc.) et appel de fonctions de l'application (primitive `call`). Implicitement, chaque exécution d'une telle séquence est précédée d'une synchronisation sur le port `START` et suivie d'une synchronisation sur le port `END`. Comme évoqué précédemment, les flots ont pour source des générateurs d'occurrences qui permettent de modéliser des sources d'interruptions matérielles (horloge ou périphérique). Une description CLARA explicite donc complètement le lien entre l'écoulement du temps, les événements de l'environnement perçus par le système de pilotage, et les fonctions logicielles qu'il exécute en réaction.

4.3. Environnement physique

CLARA ne permet pas de modéliser l'environnement physique. L'utilisation de générateurs d'occurrences permet cependant de modéliser les sources de signaux ou de données externes au système. Le choix des protocoles qui contrôlent les liens entre les générateurs externes et l'interface du système permet de modéliser différentes politiques d'interaction : interruption, scrutation, etc.

4.4. Temps

Dans une description CLARA, le temps apparaît à 3 niveaux. Premièrement, au niveau des générateurs

d'occurrences, pour spécifier les instants de production. Ensuite, dans la description du comportement d'une activité, l'architecte donne une estimation du meilleur et du pire temps d'exécution des fonctions utilisateur appelées (par exemple `calculCmd` figure 6). Elles deviennent des exigences à respecter pour la phase d'implémentation. Enfin, le temps intervient au niveau des contraintes portant sur une configuration (section `CONSTRAINTS` de la figure 8). Ces contraintes, relatives ou absolues, portent sur la durée séparant deux occurrences. Là encore, il s'agit d'exigence qui doivent guider les étapes ultérieures du développement. Concernant l'exemple, on a exprimé (cf. fig. 8) que la lecture d'une mesure par l'instance d'activité `capture` doit être suivie par l'émission d'une action par l'instance `action` dans un intervalle de 250ms (contrainte relative). On a également exprimé une contrainte absolue : la première lecture du capteur doit avoir lieu moins de 25ms après le démarrage du système (événement associé implicitement à la date 0). Pour des raisons de lisibilité, l'expression graphique des contraintes n'a pas été portée sur la fig. 5. Dans [18], nous expliquons comment vérifier la cohérence entre les différentes grandeurs relatives au sein d'une description CLARA.

4.5. Support d'exécution

CLARA n'offre pas de moyen de décrire le support d'exécution, que ce soit au niveau matériel (nombre et caractéristiques des calculateurs, des mémoires, interconnexion entre ces éléments) ou au niveau logiciel (services fournis par l'exécutif).

4.6. Modes de fonctionnement

Dans [12], E. Durand fait une proposition d'extension de CLARA pour l'expression des modes de fonctionnement. Basiquement, un mode de fonctionnement est un composant hiérarchique qui peut contenir des objects CLARA classiques et éventuellement des sous-modes de fonctionnement. Au sein du composant système, ainsi qu'au sein de chaque mode comportant des sous-modes, un réacteur est décrit, qui spécifie les conditions de commutation entre les sous-modes. Cette proposition n'a pas encore été intégrée au langage.

4.7. Outils

Le langage CLARA a pour vocation de servir de base à différents travaux académiques menés au sein de l'équipe « Systèmes Temps Réel » de l'IRCCyN. Les outils développés pour et autour de lui n'ont donc pas dépassé le stade de prototype et servent principalement de preuve de faisabilité. Ont ainsi été étudiés :

- la compilation vers les réseaux de Petri temporels pour la vérification de propriétés avec hypothèse d'architecture matérielle à ressources infinies [12] ;
- la projection d'une architecture CLARA sur un support d'exécution OSEK/VDX distribué et une technique de validation par simulation d'un modèle SDL de l'architecture opérationnelle résultante [17] ;
- l'utilisation des outils ROMÉO, TINA et CADP pour la vérification de propriétés de sûreté et la vérification de la cohérence des grandeurs temporelles (budget d'exécution vs. contraintes) [18] ;
- l'utilisation des outils de transformation de modèle ATL pour la projection d'architecture CLARA sur des supports d'exécution temps réel (VxWorks, RTAI) [10].

5. MetaH

La spécification complète de l'exemple traité est donnée en annexe A.2.

5.1. Présentation

MetaH est un langage développé au cours des années 90 par S. Vestal et son équipe chez Honeywell, dans le cadre de l'initiative « Domain Specific Software Architecture » [5]. Leurs travaux constituent une des bases de la norme AADL. Le langage permet de décrire l'architecture logicielle, l'architecture matérielle et la projection de la première sur la seconde *i.e.* l'architecture opérationnelle, à des fins de validation et de génération de code. Plus précisément, les systèmes visés sont : critiques, temps réel, complexes et multiprocesseurs.

```
process ICalculCommande is
    etat: in port ETR05.T_ETAT;
    consigne: in port ETR05.T_ETAT;
    mesure: in port ETR05.T_CMD_IN;
    commande: out port ETR05.T_CMD_OUT;
end ICalculCommande;
process implementation ICalculCommande.Default is
    calculCmd: subprogram;
paths
    <<Normal>> := calculCmd;
attributes
    calculCmd'SourceTime := 20ms;
    self'ComputePath := Normal;
end ICalculCommande.Default;
```

FIG. **9. Définition d'un type d'interface et d'implémentation de process**

Les composants principaux sont les **processes**. Un process est une unité d'ordonnancement (et optionnellement un espace d'adressage protégé). Il s'agit donc bien d'une entité de l'architecture opérationnelle, contrairement aux activités atomiques CLARA qui peuvent être projetées selon différentes stratégies (regroupement, éclatement, etc.). On commence par définir un type d'interface (cf. fig. 9), qui décrit les

ports d'entrée et de sortie du process (les variables importées et exportées), les événements qu'il peut produire et les **package** et **monitor** qu'il rend disponible (ce sont les mêmes notions que dans Ada, langage auquel MetaH est fortement lié). On peut ensuite définir différents types d'implémentations correspondant au type d'interface. Dans une implémentation, on définit les flots de contrôle du process (rubrique `paths`) ainsi que différents attributs relatifs par exemple au temps d'exécution. Dans la définition de l'implémentation d'un process, aucune information n'est obligatoire. Cela correspond à la philosophie des ADLs : une description architecturale accompagne le développement du système et se complète donc au fur et à mesure.

Lors de l'instanciation d'un process (cf. fig. 10), des informations supplémentaires peuvent être précisées. Il faut ainsi spécialiser le process en `aperiodic` ou `periodic` et le cas échéant préciser sa période. Un process apériodique est activé sur occurrence d'un événement matériel ou logiciel, tandis qu'un process périodique est activé par la couche exécutif MetaH (MetaH est un ADL « implementation dependent », qui vise une plate-forme d'exécution particulière. A contrario, CLARA et Wright sont « implementation independant »). Le comportement d'un process est défini par un ensemble de `path` qui sont des séquences finies sans point de blocage (voir paragraphe 5.2).

```
macro Etr05 is
    mesure: in port ETR05.T_MESURE;
    consigne: in port ETR05.T_ETAT;
    action: out port ETR05.T_ACTION;
end Etr05;
macro implementation Etr05.Default is
    capture: periodic process ICapture.Default;
    calculCommande: periodic process
      ICalculCommande.Default; ...
connections
    capture.mesure_in <- mesure; ...
    action <<- action.action;
    calculCommande.mesure <<- capture.mesure_out; ...
    calculCommande.etat <- majEtat.etat;
attributes
    ...
    majEtat'Period := 500ms;
    action'Period := 500ms;
    action'Deadline := 250ms;
end Etr05.Default;
```

Fig. **10. Description d'une macro**

Le composant **macro** (cf. fig. 10) permet de structurer une spécification d'architecture logicielle. Il encapsule une configuration pour en faciliter la manipulation. Il possède une clause `connections` pour spécifier les interconnexions entre composants. En ce qui concerne la signalisation, il est possible de connecter un événement à un process apériodique (déclenchement) ou à un mode (commication). En ce qui concerne les flots de données, MetaH supporte la connexion « undelayed » notée `<<-` où la donnée est rendue disponible au consommateur dès la terminaison du producteur, et la connexion « delayed » notée `<-` où la donnée est rendue disponible au consommateur uniquement à la fin de la période du producteur.

Le composant **mode** correspond à la notion de mode de fonctionnement. Un mode encapsule une configuration de composants dont l'activation est liée à celle du mode. Les commutations de mode se font sur occurrence d'événement (logiciel ou matériel).

```
application Etr05 is
    macro Etr05.Default on
      processor MPC565.Default;
connections
    Etr05.mesure <- MPC565.port1;
    Etr05.consigne <- MPC565.port2;
    MPC565.port3 <<- Etr05.action;
attributes
    MPC565'ClockPeriodMax := 64us;
end Etr05;
```

Fig. **11. Description de l'application.**

Enfin, le composant **application** (cf. fig. 11) est la racine d'un nœud du système. Il permet en de mettre en correspondance les éléments de l'architecture logicielle et ceux de l'architecture matérielle de ce nœud : un événement logiciel peut être associé à une interruption, un port logiciel peut être associé à un port matériel, etc.

5.2. Aspects réactifs

Dans MetaH, les flots de contrôle sont associés aux processu. Ils sont activés soit périodiquement, soit en réaction à un événement. Ils peuvent être suspendus lorsqu'une instance de process accède à un moniteur partagé, et stoppés suite à la terminaison d'un process ou à une commutation de mode. Le comportement d'un process est décrit via la clause `path` comme un ensemble de séquence finie d'appel à des **subprograms** et d'accès à des monitor ou package. Les données en entrée sont lues avant l'activation et les données en sortie sont publiées après la terminaison. Les flot de données interprocess induisent donc de simple contrainte de précédence entre process. Si l'on impose que chaque subprogram implante un algorithme (et donc termine) et si la spécification est complète (i.e. les clauses `paths` décrivent bien tous les chemins possibles), on déduit l'ensemble des flots de contrôle. Par conséquent, les réactions de l'application à un événement, qu'il s'agisse d'un événement produit par l'environnement (et relayé via une interruption) ou l'expiration d'une horloge, sont toutes décrites.

Par ailleurs, les contraintes imposées sur le comportement des process permettent de s'assurer que les modèles utilisés pour les analyses (en particulier d'ordonnançabilité) sont cohérents avec l'implémentation. Ce problème est traité dans [40], en utilisant une approche formelle originale (par analyse de modèles obtenus par instrumentation du code).

5.3. Environnement physique

Comme CLARA, MetaH permet d'expliciter uniquement les points de communications entre le système de contrôle et le système contrôlé par le biais des événements matériels, des ports matériels et des périphériques (cf. paragraphe 5.5).

5.4. Temps

Le temps dans MetaH est exprimé via différents attributs : période et échéance d'un process, temps d'exécution d'un path. L'attribut période se passe de commentaire. L'attribut échéance, associé à une instance de process, est le seul moyen d'exprimer des contraintes de temps. Tandis que CLARA vise à décrire l'architecture fonctionnelle des applications et doit donc offrir des mécanismes abstraits d'expression de contraintes destinées à être raffinées, MetaH décrit l'architecture opérationnelle et permet donc uniquement l'expression de contraintes primitives, manipulables directement par l'ordonnanceur (avec une politique EDF ou DM) et par les outils d'analyse d'ordonnançabilité de l'atelier construit autour du langage. Enfin, les temps d'exécution (associés aux `path` ou aux `subprogram`) sont donnés sous la forme de constantes correspondant aux pires temps d'exécution. Ils sont également destinés à être utilisé par les outils d'analyse d'ordonnançabilité.

5.5. Support d'exécution

MetaH est l'un des rares ADL qui permet la description de l'architecture matérielle des systèmes. Sans entrer dans les détails, les principaux composants disponibles pour ce faire sont : **processor** (une carte pour laquelle on spécifie le modèle de processeur, l'exécutif et la chaîne de développement associée, ainsi qu'une interface permettant la liaison entre l'architecture matérielle et l'architecture logicielle), **device** (un périphérique), **memory** (bloc de mémoire, éventuellement partagée, caractérisé par sa taille en octet et le schéma d'adressage utilisé), **channel** (bus point-à-point bidirectionnel utilisé pour interconnecter différents processor et / ou différents devices) et enfin **system** (racine de la spécification). Ces composants possèdent bien entendu des interfaces, constituées de port et d'événements, destinés à être finalement mises en correspondance avec leurs homologues logicielles. La description du support d'exécution est très peu utilisée par les outils d'analyse et permet avant tout la construction et le déploiement automatisés d'images binaires correspondant aux spécifications.

5.6. Modes de fonctionnement

Les composants logiciels mode « définissent des configurations en-ligne alternatives ». Lorsqu'un mode devient actif, la configuration qu'il englobe est mise en place (activation de process, mise à jour des tables de la couche exécutif MetaH). Inversement, lorsqu'un mode devient inactif, la configuration en cours est « détruite » (arrêt des process concernés). Des modes frères dans l'arborescence associée à l'architecture logicielle sont mutuellement exclusifs mais l'utilisation de macro permet d'avoir plusieurs modes actifs au même instant. Enfin, comme évoqué ci-avant, les commutations de mode sont provoquées par l'occurrence d'événements.

5.7. Outils

Autour de MetaH, un atelier complet existe, qui permet la validation de l'architecture proposée et la génération automatique de l'application associée. La figure 12 issue de [39] en présente la structure. Elle illustre bien l'intérêt d'un développement orienté architecture, qui permet de produire automatiquement, à partir d'une même source, les modèles de validation et l'implémentation.

Comme le précisent les auteurs, l'atelier est un prototype, toujours en développement [6]. Il a cependant été utilisé avec succès sur quelques cas d'études industriels (en particulier un système de guidage de missile [31]) et a montré sur ces expériences que l'approche architecturale, lorsqu'elle est supportée par un atelier logiciel adéquate, permet de diminuer les temps et coût de conception et développement d'un système temps réel.

6. COTRE

Destiné à proposer une méthode de conception accompagnée d'outils logiciels (de simulation, analyse d'ordonnançabilité, vérification comportementale, fiabilité, évaluation de performances, etc.) pour les systèmes avioniques, le projet COTRE[5] [16] a choisi de s'appuyer sur un ADL pour décrire les architectures de ces systèmes.

De façon à permettre le développement simultané du niveau utilisateur et du niveau vérification (plus formel), deux langages frères ont été définis : UCOTRE (pour « User COTRE ») proche de l'architecte et VCOTRE (pour « Verification COTRE ») proche des formalismes de vérification. Le but initial était de fusionner les deux branches au sein d'un unique langage. Finalement, COTRE est présenté aujourd'hui comme un profil AADL [15]. En tirant profit des mécanismes d'extension offerts par AADL, COTRE offre ainsi des constructions pour la spécification de :

– contrat, c'est-à-dire l'expression de propriétés requises par un composant (suppositions / exigences sur le comportement de l'environnement)

[5] COTRE (« COmposant Temps REel ») est un projet RNTL qui a débuté en janvier 2002 pour une durée de 2 ans et dont le consortium est composé de AIRBUS, TNI-Valiosys, ENSTB et FéRIA.

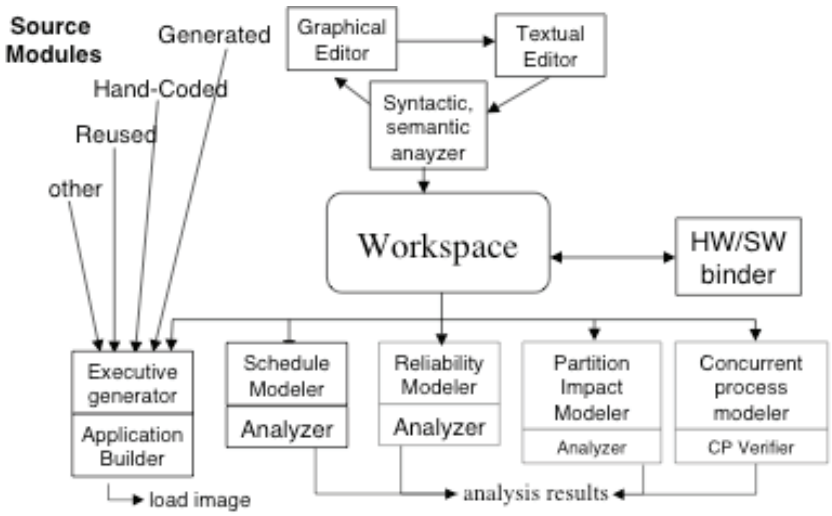


FIG. **12. L'atelier construit autour de MetaH [39]**

et de propriétés garanties par le composant (relatives à son comportement). Ces propriétés s'expriment à l'aide d'assertions de haut niveau ou d'équivalences comportementales ;
- propriétés (attributs) relatives à l'implémentation d'un composant élémentaire : période d'activation, priorité, phase, etc. ;
- comportement d'un composant sous la forme de machines de Mealy.

Les travaux dont nous avons connaissance ont porté sur l'utilisation de UCOTRE comme profil AADL. Actuellement, la cohérence sémantique entre UCOTRE et AADL, d'une part, et UCOTRE et VCOTRE, d'autre part, est à l'étude.

Le langage VCOTRE [14] est quant à lui présenté comme un langage intermédiaire, interface entre l'ADL et les différents formalismes de vérification utilisés dans le projet : systèmes de transitions, automates temporisés et réseaux de Petri. Ainsi, dans VCOTRE, suivant la philosophie des ADLs, l'architecture statique est décrite sous forme d'une structure hiérarchique de composants ; les composants présentent une interface ; leur composition utlise des connecteurs. Les composants élémentaires sont soit des processus applicatifs, soit des composants de communication et de synchronisation (sémaphore, boîte aux lettres, etc.) issus de la spécification ARINC 653. Le comportement d'un processus est décrit par un système de transitions étiquetés. L'interface est constituée d'un ensemble de ports de communication (typés) et de l'ensemble des propriétés qui forment le contrat du composant. Ces propriétés peuvent être choisies parmi un ensemble de propriétés prédéfinies ou exprimées directement à l'aide du langage d'assertions supporté par l'outil de vérification visé.

7. Conclusion et perspectives

Avec ce chapitre, notre intention est d'introduire simplement et sans ambition d'exhaustivité les ADLs et plus spécifiquement les ADLs pour le temps réel. Cette présentation doit donc être considérée comme préalable à une étude plus approfondie. Après avoir situé la notion d'architecture et les enjeux qui s'y rapportent, nous avons identifié et illustré les principaux concepts-clés des ADLs généralistes. Les caractéristiques de ces langages pouvant constituer une réponse aux problèmes posés par l'augmentation de la complexité des systèmes temps réel, nous nous sommes ensuite attardé sur les ADLs temps réel. Si les concepts de base et les abstractions correspondantes sont bien sûr à conserver, nous avons souligné que pour différentes raisons, ils devaient être adaptés et enrichis pour satisfaire aux spécificités du domaine visé. C'est ce que nous avons voulu ensuite montrer en présentant ensuite CLARA et MetaH, assortis d'un même exemple illustratif. Par manque de place, le projet COTRE a été abordé de manière succincte. Pour la même raison, nous avons passé sous silence d'autres travaux étiquetés ADL temps réel comme Unicon [42] ou traitant de la conception architecturale des systèmes temps réel comme Basement [23]. Quant à AADL, un autre chapitre lui est consacré (voir le chapitre « Les ADL du point de vue de l'industrie » dans ce volume).

D'une manière générale, il faut admettre que la multiplicité des propositions académiques en matière d'ADLs (et notamment pour le temps réel) n'a pas favorisé l'aboutissement des travaux associés tout comme leur adoption par le milieu industriel. La mise en place du langage AADL en tant que standard SAE devrait maintenant constituer un référence pour le

domaine temps réel. Par les mécanismes d'extension qu'il offre, son adaptation locale est possible. C'est d'ailleurs la voie retenue par le projet COTRE et il nous semble qu'il faille tendre maintenant vers une telle unification.

Nous pensons que l'ADL et la description d'architecture ne doivent pas être cantonnés à la seule étape de conception architecturale. Ainsi, dans le cas de MetaH, la description architecturale (et les attributs qu'elle porte) est le point de départ pour la génération d'une implémentation mais aussi pour la dérivation des divers modèles exploités par les outils d'analyse. Pour réduire davantage la cassure sémantique entre les modèles de V&V et l'implantation, ou du moins pour faciliter la traçabilité entre ces niveaux, il faut penser l'ADL pour qu'il constitue « l'épine dorsale » des différentes étapes du processus de développement. D'une manière schématique, une étape extrait de la description d'architecture alors disponible les informations qui la concernent, puis, en retour, l'enrichit des données qu'elle a produites. Un tel retour d'informations au niveau de la description simplifie l'interprétation des résultats et peut permettre à terme une utilisation transparente des techniques formelles.

Des travaux en matière d'outillage d'ADL doivent bien sûr être poursuivis afin d'assister l'architecte tant sur le plan de la V&V que du déploiement. Ainsi, sur ce dernier point, la tâche est plus ou moins difficile selon le degré de couplage qu'affiche l'ADL avec les aspects opérationnels. Dans le cadre de CLARA, le niveau abstrait choisi pour décrire une architecture fait que les décisions relatives au déploiement restent entièrement à prendre : il s'agit bien alors de « construire » et non de « traduire » une implémentation. Pour un système temps réel, il s'agit schématiquement de définir les tâches, les trames, leur placement sur les ressources de calcul et de communication, la configuration résultante des supports d'exécution, etc. Pour une même architecture, l'espace des implémentations peut donc être important et des outils d'exploration et d'aide à la décision doivent être proposés. Signalons que le couplage outils de déploiement et langages de niveau conception (même s'ils ne s'agit pas à proprement parler des ADLs) est au cœur de la problématique « Model Integrated Computing » [38].

Enfin, dans la mesure où les problèmes ici soulevés sont difficiles à traiter, il apparaît incontournable de les traiter dans un cadre bien délimité et restrictif. L'identification de classes d'applications, de familles d'architectures, de patrons architecturaux, ou encore de styles architecturaux contribue à une plus grande spécialisation des langages et modèles utilisés et favorise la mise en place de schémas de conception éprouvés. À notre connaissance, il s'agit d'un point encore peu abordé dans le domaine des ADLs temps réel.

Références

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Conference on Fundamental Approaches to Software Engineering (Lisbonne, Portugal)*, 1998.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. Thèse de doctorat, School of Computer Science, Carnegie Mellon University, 1997.

[3] M. Barbacci et al. Durra : a structure description language for developing distributed applications. *Software Engineering Journal*, 8(2) :83–94, 1993.

[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, 1998.

[5] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2), 1996.

[6] P. Binns and S. Vestal. Hierarchical Composition and Abstraction in Architecture Models. In Dissaux et al. [11], pages 35–50.

[7] P. Clements. Attractions in software architecture. Technical Report CMU/SEI-96-TR-008, Software Engineering Institute, 1996.

[8] F. de Remer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2), 1976.

[9] V. Debruyne, F. Simonot-Lion, and Y. Trinquet. EAST-ADL : an Architecture Description Language. Validation and Verification aspects. In Dissaux et al. [11], pages 181–196.

[10] J. Delatour, F. Thomas, G. Savaton, and S. Faucou. Modèle de plate-forme pour l'embarqué : première expérimentation sur les noyaux temps réel. In *Ingénierie Dirigée par les Modèles*, 2005.

[11] P. Dissaux, M. Filali Amine, P. Michel, and F. Vernadat, éditeurs. *Architecture Description Languages*, volume 176 of *IFIP*. Springer, 2004.

[12] E. Durand. *Description et vérification d'architecture temps réel :* Clara *et les réseaux de Petri temporels*. Thèse de doctorat, École Centrale de Nantes, Nantes, France, 1998.

[13] A. Déplanche et al. Rapport de synthèse – AS 195 - Composants et Architectures Temps réel. Technical Report RI2005_1, IRCCyN, 2005.

[14] P. Farail et al. The COTRE project : how to model and verify real-time architecture ? In *2nd European Congress on Embedded Real-Time Software (ERTS'2004)*, Toulouse, 2004.

[15] P. Farail and P. Gaufillet. COTRE as an AADL profile. In Dissaux et al. [11], pages 167–179.

[16] J. Farines et al. The COTRE project : rigirous software development for real-time systems in avionics. In *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, mai 2003.

[17] S. Faucou. *Description et construction d'architectures opérationnelles validées temporellement*. Thèse de doctorat, Université de Nantes, Nantes, France, 2002.

[18] S. Faucou, A. Déplanche, and Y. Trinquet. An ADL-centric approach for the formal design of real-time systems. In Dissaux et al. [11], pages 67–82.

[19] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR2 User Manual*, 2003.

[20] D. Garlan. Software Architecture : a Roadmap. In A. Finkelstein, éditeur, *International Conference on Software Engineering - Proceedings of the conference on the Future of Software Engineering*, pages 91–101. ACM Press, 2000.
[21] D. Garlan, R. Monroe, and D. Wile. ACME : An Architecture Description Interchange Language. In *CASCON'97*, pages 169–183, novembre 1997.
[22] D. Garlan and M. Shaw. *An introduction to software architecture*. World Scientific Publishing, 1993.
[23] H. Hansson, H. Lawson, and M. Strömberg. Basement : a distributed real-time architecture for vehicle applications. *Real-Time Systems*, 11(3) :223–244, 1996.
[24] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
[25] IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - ANSI/IEEE Standard 147-2000, 2001.
[26] J. Kramer, J. Magee, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering Software*, 15(6) :663–675, 1989.
[27] P. Kruchten. Architectural blueprints - the "4+1" view model of software architecture. *IEEE Software*, 12(6), 1995.
[28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *European Software Engineering Conference*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.
[29] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, octobre 1996.
[30] R. Marvie. *Les intergiciels*, chapitre Langages de description d'architectures - Un état de l'art. Hermès, 2005.
[31] D. McConnell, B. Lewis, and L. Gray. Reengineering a single threaded embedded missile application onto a parallel processing platform using metah. *Real-Time Systems*, 14(1) :7–20, 1998.
[32] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Language. *IEEE Transactions on Software Engineering*, 26(1) :70–93, Jan. 2000.
[33] P. Oreizy, N. Medvidovic, and R. N. Talylor. Architecture-Based Runtime Software Evolution. In *International Conference on Software Engineering*, pages 177–186. IEEE Computer Society, 1998.
[34] F. Paulisch. Software architecture and reuse : an inherent conflit ? In *3rd International Conference on Software Reuse*, 1994.
[35] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.
[36] R. Prieto-Diaz and J. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4) :307–334, 1986.
[37] Society of Automotive Engineer. *AS5506 - Architecture Analysis and Design Language (AADL)*, 2004.
[38] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4) :110–111, 1997.
[39] S. Vestal. *MetaH User's Manual - version 1.27*. Honeywell Technology Center, Minneapolis, MN, USA, 1998.
[40] S. Vestal. Modeling and Verification of Real-Time Software Using Extended Linear Hybrid Automata. In *Lfm2000 : Fifht NASA Langley Formal Method Workshop*, pages 83–94, 2000.
[41] A. Wall. *Architectural modeling and analysis of complex real-time systems*. Thèse de doctorat, Department of Computer Science and Engineering, Mälardalen University, 2003.
[42] G. Zelesnik. The UniCon Language Reference Manual. Technical report, School of Computer Science, Carnegie Mellon University, 1996.

A. L'exemple (presque) complet

A.1. En CLARA

```
TYPE_ACTIVITY T_Capture;
    VAR_IN mesure_in : t_mesure;
    VAR_OUT mesure_out : t_cmd_in;
    BEHAVIOR receive(mesure_in)
        -> call(InToOut, 100us, 500us)
        -> send(mesure_out);
END_ACTIVITY.
TYPE_ACTIVITY T_CalculCommande;
    VAR_IN etat, consigne : t_etat;
        mesure : t_cmd_in;
    VAR_OUT commande : t_cmd_out;
    BEHAVIOR receive(mesure) -> receive(etat)
        -> receive(consigne)
        -> call(calculCmd, 10ms, 20ms)
        -> send(commande);
END_ACTIVITY;
TYPE_ACTIVITY T_MajEtat;
    VAR_IN commande : t_cmd_out;
    VAR_OUT etat : t_etat;
    BEHAVIOR receive(commande)
        -> call(calculEtat, 100ms, 250ms)
         -> send(etat);
END_ACTIVITY;
TYPE_ACTIVITY T_Action;
    VAR_IN commande : t_cmd_out;
    VAR_OUT action : t_action;
    BEHAVIOR receive(commande)
        -> call(cmdToAction, 0.5ms 1ms)
        -> send(action);
END_ACTIVITY;
TYPE_GENERATOR T_Horloge_500ms;
    PERIODIC 500ms;
    SIGNAL_OUT out;
    BEHAVIOR CLOCK;
END_GENERATOR;

SYSTEM Etr05;
VAR_IN mesure : t_mesure; consigne : t_etat;
VAR_OUT action : t_action;
GENERATOR H1 : T_Horloge_500ms;
ACTIVITY
    capture : T_Capture;
    calculCommande : T_CalculCommande;
    majEtat : T_MajEtat;
    action : T_Action;
LINKS
    // liens d'interface
    mesure TO capture.mesure_in;
    consigne TO calculCommande.consigne;
    action.action TO action;
    // liens inter-activites
    capture.mesure_out TO
        calculCommande.mesure : RDVd;
    calculCommande.commande TO
        (action.commande : RDVd
        & majEtat.commande : BAL);
    majEtat.etat TO calculCommande.etat : RAFO;
    // liens d'activation
    H1.out TO capture.start : MEM;
```

```
    calculCommande.end TO calculCommande.start : MEM;
    action.end TO action.start : END;
    majEtat.commande TO majEtat.start : INC;
CONSTRAINTS
    Abs(capture.mesure_in.cnf) < 25ms;
    480ms < capture.mesure_in.cnf < 520ms;
    0ms < capture.mesure_in.cnf :
          action.action.cnf     < 250ms;
END_SYSTEM.
```

A.2. En MetaH

```
with type package ETR05;

-- declaration des interfaces des processus
process ICapture is
    mesure_in: in port ETR05.T_MESURE;
    mesure_out: out port ETR05.T_CMD_IN;
end ICapture;
process ICalculCommande is
    etat: in port ETR05.T_ETAT;
    consigne: in port ETR05.T_ETAT;
    mesure: in port ETR05.T_CMD_IN;
    commande: out port ETR05.T_CMD_OUT;
end ICalculCommande;
process IMajEtat is
    commande: in port ETR05.T_CMD_OUT;
    etat: out port ETR05.T_ETAT;
end IMajEtat;
process IAction is
    commande: in port ETR05.T_CMD_OUT;
    action: out port ETR05.T_ACTION;
end IAction;

-- declaration des implementations des processus
process implementation ICapture.Default is
    InToOut: subprogram;
paths
    <<Normal>> := inToOut;
attributes
    inToOut'SourceTime := 500us;
    self'ComputePath := Normal;
end ICapture.Default;
process implementation ICalculCommande.Default is
    calculCmd: subprogram;
paths
    <<Normal>> := calculCmd;
attributes
    calculCmd'SourceTime := 20ms;
    self'ComputePath := Normal;
end ICalculCommande.Default;
process implementation IMajEtat.Default is
    calculEtat: subprogram;
paths
    <<Normal>> := calculEtat;
attributes
    calculEtat'SourceTime := 250ms;
    self'ComputePath := Normal;
end IMajEtat.Default;
process implementation IAction.Default is
    cmdToAction: subprogram;
paths
    <<Normal>> := cmdToAction;
attributes
    cmdToEtat'SourceTime := 1ms;
    self'ComputePath := Normal;
end IAction.Default;

-- declaration de la macro englobant
-- l'architecture logicielle
macro Etr05 is
    mesure: in port ETR05.T_MESURE;
    consigne: in port ETR05.T_ETAT;
    action: out port ETR05.T_ACTION;
end Etr05;
macro implementation Etr05.Default is
    capture: periodic process ICapture.Default;
    calculCommande: periodic process
        ICalculCommande.Default;
    majEtat: periodic process IMajEtat.Default;
    action: periodic process IAction.Default;
connections
    capture.mesure_in <- mesure;
    calculCommande.consigne <- consigne;
    action <<- action.action;
    calculCommande.mesure <<- capture.mesure_out;
    action.commande <<- calculCommande.commande;
    majEtat.commande <<- calculCommande.commande
    calculCommande.etat <- majEtat.etat;
attributes
    capture'Period := 500ms;
    calculCommande'Period := 500ms;
    majEtat'Period := 500ms;
    action'Period := 500ms;
    action'Deadline := 250ms;
end Etr05.Default;

-- liaison avec l'archi matérielle
application Etr05 is
    macro Etr05.Default on processor MPC565.Default;
connections
    Etr05.mesure <- MPC565.port1;
    Etr05.consigne <- MPC565.port2;
    MPC565.port3 <<- Etr05.action;
attributes
    MPC565'ClockPeriodMax := 64us;
end Etr05;
```

Les ADL du point de vue de l'industrie

Jean-François Tilman
Axlog Ingénierie
19-21 rue du 8 mai 1945
94110 Arcueil, France
Jean-Francois.Tilman@axlog.fr

Résumé

Les systèmes embarqués temps réel sont de plus en plus complexes, et les exigences en terme de sûreté et de fiabilité de plus en plus fortes. Pour répondre à ce défi l'industrie développe de nouveaux processus d'ingénierie, plus formels et basés sur l'utilisation de langages de description d'architectures (ADL).

AADL est un de ces ADL, dédié aux systèmes embarqués en vue de leur conception et de leur analyse. Standardisé dans le cadre du SAE, ce langage apparaît comme un support intéressant pour l'industrie grâce à sa sémantique forte et sa flexibilité.

Plusieurs projets de recherche terminés, en cours ou en gestation illustrent l'intérêt de l'industrie pour les ADL comme base de nouveaux processus de développement. Nommons par exemple Topcased ou Assert dans le domaine avionique, ou EAST-AEE pour l'automobile.

1. Introduction

Les langages de description d'architectures (ADL, *architecture description languages*) sont de plus en plus présents dans le paysage du génie logiciel et système. Comme leur nom l'indique, ce sont des langages informatiques capables de décrire d'une manière plus ou moins formelle des architectures logicielles, électroniques ou systèmes.

Les premiers ADL sont apparus dans les laboratoires de recherche. Aujourd'hui, on note de plus en plus de projets de recherche menés par les industriels et faisant appel à de tels ADL. Cette évolution traduit certainement une maturité plus grande de ces concepts et un intérêt pour répondre à des besoins rencontrés dans les processus de développement.

Nous verrons tout d'abord les besoins qu'a l'industrie aujourd'hui pour développer ses systèmes embarqués, et qui peuvent être à l'origine de l'engouement pour les ADL. Nous présenterons ensuite un ADL particulier, AADL, qui est assez représentatif de ceux que l'on voit émerger actuellement. Nous verrons en quoi un tel ADL répond au besoins de l'industrie. Nous finirons par la présentation de quelques projets qui illustrent la place grandissante que prennent les ADL dans les processus industriels, et le passage du stade de recherche pure au stade d'industrialisation.

2. Besoins de l'industrie

2.1. Gérer la complexité

Les systèmes embarqués produits par l'industrie ont à prendre en charge un nombre croissant de fonctions, cette évolution conduisant à une complexité toujours plus grande. Une bonne illustration de ce phénomène est fournie par le monde automobile, où l'on peut voir maintenant plusieurs dizaines de calculateurs embarqués dans un véhicule. On constate aussi que les fonctions introduites ou en passe de l'être sont de plus en plus critiques : l'approche *X-by-wire* vise à contrôler tant la direction que le freinage par l'électronique plutôt que par la seule mécanique, ce qui exige de ces systèmes d'être particulièrement fiables. Dans le même temps, on note une augmentation du volume du logiciel par rapport au matériel. Ainsi l'Airbus A300B comportait 25 ko de logiciel en 1974, on en compte 64 Mo sur l'A380 en 2005 ; Spot 1 en avait 48 ko en 1980 et Mars Express en a 1,2 Mo en 2003.

Cette imbrication du logiciel et du matériel peut conduire à des échecs si elle est mal maîtrisée, et en particulier si elle n'est pas prise en compte au niveau du système lui-même. L'exemple du premier vol d'Ariane 5 l'a montré, il ne suffit pas de réutiliser des éléments ayant fait leur preuve dans un contexte donné pour qu'ils soient encore pertinants dans un autre. La gestion de la complexité passe donc par une prise en compte globale du système, et pour ce faire il y a besoin d'un moyen de rassembler et d'exploiter les informations décrivant les différents sous-ensembles, en particulier le logiciel et le matériel. Avec un tel moyen il devient possible de mettre au point des techniques et des outils vérifiant plus formellement la cohérence du système.

2.2. Maîtriser la qualité

Avec l'augmentation de leur complexité, les systèmes auraient naturellement tendance à devenir moins fiables parce que plus difficiles à maîtriser. Or, on exige non seulement d'atteindre un niveau de fiabilité égal aux ni-

veaux obtenus sur les systèmes antérieurs, mais on demande même de progresser encore.

Pour atteindre ces objectifs les industriels considèrent de plus en plus l'introduction de méthodes formelles et de preuves dans leurs processus de développement. Ces approches permettent de vérifier au plus tôt une partie des propriétés du système. Les preuves, en particulier, économisent une partie de l'effort de test tout en apportant une plus grande confiance dans les résultats. On constate cependant que les méthodes formelles sont principalement utilisées à un niveau assez bas du cycle de développement. Elles servent en particulier au développement de certaines briques logicielles bien définies.

Si les méthodes formelles ont « fait leurs preuves » pour le développement d'éléments du système, on constate toujours des difficultés lors de l'intégration du système complet. À ce niveau aussi de telles méthodes devraient aider à garantir une meilleure architecture intégrant les différents composants matériels et logiciels. Ceci passe par un moyen de décrire formellement l'architecture du système, pour disposer d'un cadre sans rupture depuis la spécification du système jusqu'à l'implantation embarquée. Les besoins de traçabilité et de communication entre partenaires du projets seront alors satisfaits et la qualité globale améliorée.

2.3. Contraintes de coûts et de délais

Une autre préoccupation de l'industrie est d'atteindre ces objectifs sans faire exploser les budgets de développement. Il ne suffit donc pas de multiplier les efforts de test, qui peuvent déjà représenter jusqu'à 80 % du développement, pour assurer la qualité des systèmes complexes. Il faut au contraire repenser les processus et les appuyer sur de nouvelles approches.

La maîtrise des coûts passe par la réutilisation de développements existants. Ceci ne peut se faire que si cette réutilisation est elle-même correctement gérée au niveau du système de façon à garantir que les hypothèses de fonctionnement de chaque bloc réutilisé sont remplies, et donc qu'aucune incompatibilité n'apparaîtra entre les composants une fois intégrés.

La réutilisation passe aussi par une meilleure interopérabilité entre composants, en particulier entre logiciel et matériel. L'objectif est de permettre une conception flexible du système avec intégration de composants interchangeables, même dans le cas où ils proviendraient de fournisseurs différents.

Pour de nombreux domaines industriels le marché est aujourd'hui tendu et exige d'être très réactif. Ceci impose de minimiser le délai séparant le lancement d'un projet et sa mise sur le marché. Ici encore, l'automobile fournit un bon exemple, avec des durées de développement des véhicules de plus en plus courtes. La réponse à cette exigence passe aussi par une amélioration du processus d'ingénierie, qui permette d'accélérer les étapes du développement sans pour autant perdre en fiabilité et en qualité.

3. Présentation d'AADL

3.1. Choix du langage

Les langages que l'on peut qualifier de langages de description d'architectures, ou ADL, sont nombreux. Cet acronyme est en effet largement utilisé pour désigner les langages informatiques permettant de décrire des architectures, qu'elles soient logicielles, électroniques... ou même dans le secteur du bâtiment.

Pour répondre aux besoins de l'industrie décrits précédemment, nous ne considérerons que les ADL capables de décrire l'architecture de systèmes embarqués, et particulièrement l'interface entre le logiciel et le matériel. Cette restriction limite beaucoup le nombre d'ADL disponibles. Certains sont spécialisés pour des besoins métier spécifiques, comme AIL-transport et EAST-ADL [7] pour l'automobile. D'autres, bien qu'issus d'un domaine particulier, sont d'usage plus général ; c'est le cas pour MetaH [10] et AADL [9] développés par l'industrie aéronautique et spatiale. C'est sur ce dernier langage que nous allons nous appuyer pour la suite de l'exposé parce qu'il présente plusieurs avantages : sémantique forte, généricité, extensibilité, standardisation, etc.

L'objet de cet article n'étant pas d'enseigner AADL mais d'illustrer l'intérêt des ADL, seuls les principaux concepts sont présentés. On se reportera au standard [9] ou à d'autres sources pour de plus amples informations [1, 6].

3.2. Origine d'AADL

AADL (*Architecture analysis & design language*) est un langage de description d'architectures embarquées. Il est défini et standardisé par un comité international sous l'autorité du SAE (*Society of automotive engineers*). Ce comité rassemble des industriels américains et européens venant principalement du secteur aéronautique et spatial (Honeywell, Rockwell Collins, Airbus, Dassault Aviation, ESA...).

L'histoire d'AADL commence avec MetaH, en 1991. MetaH est à la fois un ADL et une suite d'outils de conception et de développement supportant cet ADL. Il a été créé par Honeywell pour l'AMCOM (US Army). En 2001 ses auteurs décident de faire évoluer le langage et d'en faire un standard international, passage obligé pour en promouvoir l'utilisation à une large échelle.

La première version officielle d'AADL a été publiée en novembre 2004. Des annexes au standard sont actuellement en préparation et une deuxième version du langage est déjà prévue.

3.3. Description par composants

La description AADL d'une architecture de système est organisée sous la forme d'une arborescence de composants et de relations entre ces composants. La description d'un composant se fait en deux étapes.

Le *type* du composant représente l'interface visible de l'extérieur (ports d'entrée/sortie, propriétés, etc.). Un mécanisme d'héritage permet de définir un type comme étant

une extension d'un autre type.

L'*implémentation* du composant représente son contenu (sous-composantes, d'autres propriétés, etc.). Il est possible de définir plusieurs implémentations pour un même type. Il est aussi possible d'étendre une implémentation en une autre, de la même manière que pour les types.

3.4. Catégories de composants

Chaque composant appartient à l'une des dix catégories prédéfinies par le standard. Ces catégories représentent des éléments de la plate-forme d'exécution, du logiciel ou des éléments mixtes. Chaque catégorie est décrite de manière précise avec des règles de composition strictes et une sémantique forte.

Donnée composant logiciel représentant une donnée du code source. Cette donnée peut être partagée ;

Sous-programme composant logiciel représentant un point d'entrée dans le code source. Des appels aux sous-programmes peuvent être utilisés par exemple dans les *threads* ;

Thread composant logiciel représentant un flux séquentiel de commandes qui exécute des instructions. Un *thread* s'exécute dans l'espace mémoire virtuel d'un processus ;

Groupe de threads composant permettant d'organiser des *threads* d'un même processus dans des groupes logiques ;

Processus composant logiciel représentant un espace mémoire virtuel dans lequel s'exécutent un ou plusieurs *threads* ;

Processeur abstraction représentant un élément matériel *et* logiciel de la plate-forme d'exécution, responsable de l'ordonnancement et de l'exécution de *threads* ;

Mémoire composant de la plate-forme d'exécution capable de contenir des images binaires (données, sous-programmes, processus) ;

Bus composant de la plate-forme d'exécution qui peut échanger des commandes et des données entre mémoires, processeurs et périphériques ;

Périphérique *(device)* composant de la plate-forme d'exécution qui sert d'interface avec l'environnement extérieur (capteurs et actionneurs) ;

Système composant mixte représentant l'assemblage de composants logiciels, de composants de la plate-forme d'exécution et d'autres composants systèmes.

L'exemple suivant présente la description en AADL de la charge utile d'un satellite avec un jeu de caméras, des processeurs pour traiter les données, une mémoire de masse et un bus de communication. La description de l'implémentation se fait ici en deux étapes, la deuxième servant à raffiner la première. On notera également que l'ordre des déclarations n'importe pas et qu'un élément peut être utilisé avant d'être décrit.

```
-- Description d'un type vide de charge utile
system payload end payload ;

-- Description d'une première implémentation
-- avec quelques sous-composants
system implementation payload.proto
   subcomponents
      camera_set : system ;
      main_processor : processor ;
      signal_processor : processor ;
      payload_bus : bus ;
end payload.proto ;

-- Seconde implémentation surchargeant la
-- première en complétant et précisant la liste
-- des sous-composant.
system implementation payload.newproto
      extends payload.proto
   subcomponents
      camera_set : refined to
         system camera_system ;
      main_processor : refined to
         processor sparc.leon ;
      signal_processor : refined to
         processor dsp ;
      payload_bus : refined to
         bus spacebus ;
      mass_memory : memory ;
end payload.newproto ;

-- Description des types et implémentations
-- nécessaires aux sous-composants.
system camera_system end camera_system;
bus spacebus end spacebus ;
processor dsp end dsp ;
processor sparc end sparc ;
processor implementation sparc.leon
end sparc.leon ;
```

3.5. Caractéristiques

Outre les composants, AADL propose d'autres éléments pour représenter les *caractéristiques (feature)* d'un type de composant, et qui entrent dans la description de ce dernier. Ces caractéristiques sont les ports d'entrées/sorties, les sous-programmes mis à disposition par le composants, les paramètres de sous-programmes, les accès à des sous-composants. Elles peuvent être reliées par des connexions, que l'on décrit dans l'implémentation du composant.

3.6. Propriétés

À tous ces éléments du langage on peut associer des propriétés. Une propriété permet d'affiner et de paramétrer une description. Certaines sont prédéfinies par le standard (p. ex. la période d'activation d'un processus périodique ou la taille d'une mémoire). D'autres peuvent être créées et ajoutées librement par l'utilisateur pour répondre à des besoins qui lui seraient propres.

On peut ainsi imaginer des propriétés qui exprimeraient la masse et que l'on associerait à tous les composants autres que logiciels. L'exemple ci-dessous montre comment ceci s'écrit en AADL. Il complète l'exemple précédent et introduit la description et l'utilisation de nouvelles propriétés.

```
-- Définition d'un nouveau jeu de propriétés.
property set space_properties is
   -- un type de propriété masse
   mass_t: type aadlreal
           units space_properties::mass_u ;
   -- une unité de masse
   mass_u: type units
      (g, kg => g*1000, t => kg*1000);
   -- un type de propriété interval de masses
   mass_range_t: type range of
                 space_properties::mass_t;
   -- propriété donnant la masse autorisée
   allowed_mass: space_properties::mass_range_t
      applies to (memory, processor, bus,
                  device, system);
   -- propriété donnant la masse réelle
   actual_mass: space_properties::mass_t
      applies to (memory, processor, bus,
                  device, system);
end space_properties ;

-- nouveau type de charge utile avec une
-- indication de masse autorisée.
system payload2 extends payload
   properties
      space_properties::allowed_mass
      => constant 200.0kg .. 500.0kg ;
end payload2 ;

-- nouvelle implémentation de charge utile
-- avec une indication de masse réelle.
system implementation payload2.massive
   extends payload.newproto
   properties
      space_properties::actual_mass
         => 300.0 kg ;
end payload2.massive ;
```

3.7. Extensibilité

Nous venons de voir que les propriétés que l'on associe aux éléments du langage peuvent être enrichies librement. Ce mécanisme est une première manière d'étendre les capacités de description d'AADL pour couvrir des besoins spécifiques de l'utilisateur.

Lorsque les propriétés ne suffisent plus, AADL propose l'introduction d'*annexes*. Il s'agit de blocs de description repérés par une balise particulière et pouvant contenir n'importe quel sous-langage. Ces blocs sont par défaut simplement ignorés par tout outil AADL, ce qui permet de ne pas rendre invalide une telle description si la syntaxe de l'annexe n'est pas connue. Bien entendu l'utilisateur qui introduit une telle annexe a aussi la responsabilité de la traiter lui-même avec ses propres outils, puisque les outils standard ne la connaîtront pas.

3.8. Usages d'AADL

Disposer d'un moyen de description tel qu'AADL permet d'envisager de nombreuses utilisations au fur et à mesure de l'avancement du développement. Pour commencer, AADL permet d'assurer par construction un certain niveau de cohérence du système. Ainsi, il n'est pas permis d'insérer un composant matériel comme sous-composant d'un composant logiciel. Même si ce genre de considération est évident, nombre de langages de modélisation n'ont pas la sémantique suffisante pour l'interdire.

Dans la mesure où l'on dispose des outils adaptés, des vérifications complémentaires peuvent être menées sur la description du système. On peut par exemple vérifier si les tâches sont ordonnançables, si les temps de traitement et de transfert des données sont compatibles avec les exigences de performance, si les ressources sont correctement dimensionnées, si l'on a bien associé chaque *thread* à un processeur pour l'exécuter, etc.

À partir d'une telle description, on peut aussi générer automatiquement du code. MetaH, en son temps, reposait sur ce principe et permettait un portage aisé et rapide d'un code source de l'utilisateur sur différents systèmes embarqués ; le principe étant de générer automatiquement la « glue logicielle » qui relie et gère les fonctions utilisateur, ces dernières étant simplement données par leur code source. La jeunesse du standard AADL n'a cependant pas encore permis à de tels outils, dédiés à ce langage, de faire leur apparition sur le marché.

Les capacités d'extension d'AADL permettent d'aller beaucoup plus loin dans l'abstraction. On peut utiliser ce langage plus en amont dans le cycle de développement, par exemple pour supporter la capture d'exigences fonctionnelles ou non fonctionnelles, l'utilisation de preuves, etc.

3.9. Points forts

S'il fallait résumer les points forts d'AADL pour en justifier l'utilisation dans notre contexte, nous pourrions retenir ceux-ci :

Niveau système AADL permet la description d'architecture en prenant en compte à la fois le logiciel et la plate-forme d'exécution, alors que beaucoup d'ADL sont plus spécialisés dans l'un ou l'autre de ces aspects ;

Standardisation Le fait qu'AADL soit standardisé au niveau international permet de diffuser une version commune et unique du langage, sans éparpillement en une multitude de solutions incompatibles. Ceci donnera confiance aux vendeurs d'outils pour y investir et étoffer l'offre autour de ce langage ;

Sémantique forte Contrairement à d'autres langages, UML par exemple, AADL donne un sens précis à chacun des éléments qu'il propose. Ceci autorise des vérification de cohérence des descriptions ;

Flexibilité et extensions Les capacités d'extension d'AADL permettent de l'adapter à des usages particuliers sans perdre le bénéfice des points forts du langage ;

Acquis S'appuyant sur MetaH et plus de dix ans d'expérience, AADL est une solution crédible pour décrire les architectures de systèmes embarqués.

4. Apport des ADL pour l'industrie

En quoi les langages de description d'architectures tels qu'AADL constituent-ils un élément de solution aux besoins de l'industrie vus précédemment ?

4.1. Description formelle du système

Ces langages fournissent un moyen de décrire à la fois le logiciel embarqué et la plate-forme d'exécution sur laquelle fonctionnera le logiciel. L'adoption de ce point de vue système permet d'éviter une séparation trop profonde entre logiciel et matériel, ce qui se traduirait inévitablement par des difficultés lors de l'intégration.

Différents langages permettaient déjà de décrire des composants matériels ou logiciels – par exemple Lustre ou Esterel. Ces derniers ont montré l'intérêt qu'il y a à disposer d'une description formelle : Elle permet de supporter des méthodes formelles et des outils traitant de manière prouvée ou certifiée ces descriptions pour générer du code ou mener des vérifications.

Les ADL qui portent sur le système complet permettent donc de considérer l'ensemble des éléments du système, potentiellement développés avec des méthodes formelles spécifiques, et de décrire leur intégration et leurs relations. Ils comblent ainsi le manque de liens entre éléments logiciels et matériels.

De la même manière que pour les langages et méthodes formelles dédiées au logiciel, on conçoit bien qu'une description seule n'est pas suffisante. Les ADL ne seront utiles que comme support de méthodes et processus s'appuyant sur eux pour développer les systèmes. C'est ici que devient importante la clarté de la sémantique que les ADL donnent aux notions manipulées. En effet, une méthode formelle ne pourra pas s'appuyer sur une description floue. Il est indispensable qu'il n'y ait qu'une interprétation possible d'une description donnée, et qu'elle soit suffisamment riche pour qu'on puisse en déduire quelque chose. Une bonne illustration de ce besoin est donnée par UML. À la base, ce langage n'offre que des boîtes et des relations entre ces boîtes. Pour être capables d'aller plus loin dans son exploitation, nombre d'outils UML ont besoin de la notion de *profile*, qui restreint l'utilisation d'UML et précise un peu plus le sens des éléments mis en jeu.

4.2. Outils

Il n'est pas réaliste d'introduire une nouvelle méthode de génie système dans l'industrie si elle n'est pas supportée par des outils, quand bien même serait-elle idéale pour résoudre nombre des problèmes évoqués plus haut. Les ingénieurs du métier n'étant pas des experts de ces méthodes théoriques, les outils sont là pour les guider et les décharger de toutes les tâches fastidieuses.

Les outils informatiques ont besoin de descriptions adaptée à leurs capacités de traitement. Les ADL sont là pour ça, dans la mesure où ce sont des langages informatiques et non des langages naturels. Ils permettent de traduire dans un format exploitable par logiciel les informations que l'utilisateur peut avoir à l'esprit ou qui peuvent apparaître dans des documents de spécification ou de conception.

Au même titre que les méthodes formelles, les outils ont besoin d'une sémantique forte pour pouvoir tirer quelque chose d'utile des descriptions qu'ils manipulent. Ici encore les ADL sont utiles.

À partir du moment où l'on peut réaliser des outils, une automatisation plus large du cycle de développement devient envisageable, avec tous les avantages que cela représente : gain de temps, moins de risques d'erreurs, vérifications automatiques, etc.

4.3. Réutilisation

Nous avons vu que la réutilisation de composants déjà développés était une voie intéressante pour limiter les développements, réduire les coûts et améliorer la qualité. Pouvoir décrire complètement un composant avec toutes les informations nécessaires à sa réutilisation permet de constituer des bibliothèques de composants réutilisables. Étant des langages informatiques, les ADL se prêtent bien à cet usage.

5. Illustration par quelques projets

De nombreux projets de recherche sont menés par les industries pour définir de nouveaux processus de développement des systèmes embarqués, et plusieurs font le choix de s'appuyer sur un ADL.

Les quelques projets présentés ci-dessous sont assez représentatifs de ce courant, et d'une envergure assez large pour avoir des retombées significatives à l'avenir.

5.1. Assert

Assert est un projet intégré européen relevant du 6^{e} programme cadre de recherche et développement de la Commission européenne [2]. Il rassemble 29 partenaires, principalement dans le domaine aéronautique et spatial.

Le but de ce projet est de définir un processus amélioré de développement système et logiciel pour les systèmes embarqués temps réel critiques. Il s'appuie sur la méthode PBSE (*proof-based system engineering*), dans laquelle l'utilisation de preuves se fait de manière continue tout au long du cycle de développement, depuis la capture des exigences jusqu'à l'intégration, en passant par la spécification du système et son dimensionnement.

AADL a été choisi comme langage de base pour décrire l'architecture du système développé. Les capacités d'extension du langage sont mises à contribution pour y introduire le support de toutes les informations nécessaires, et en particulier les données spécifiques à PBSE (description des preuves, des exigences non fonctionnelles, des conditions de faisabilité, etc.).

Grâce à ce moyen de description les composants spécifiés et prouvés corrects peuvent être conservés avec leur

notice technique PBSE et réutilisés lors de nouveaux projets tout en garantissant que l'intégration est faisable.

5.2. Cotre et Topcased

Cotre et Topcased sont deux projets d'Airbus qui illustrent l'intérêt de l'industrie pour les langages de description d'architectures dans le cadre de développement de systèmes embarqués.

Cotre est un projet mené dans le cadre de l'appel RNTL 2001. Son objectif est de « *définir une démarche outillée pragmatique de modélisation et de validation d'architecture de logiciels temps réel permettant de combiner les approches formelles et semi-formelles dans un processus industriel garantissant la continuité/traçabilité de la phase de conception jusqu'à celle d'implantation du logiciel sur la cible réelle* » [3]. Il utilise AADL comme base pour décrire l'architecture et propose une extension pour supporter des informations comportementales. Ce travail est à la base des travaux actuels du comité de standardisation AADL pour définir une annexe comportementale standard.

Le projet Topcased prolonge et élargit le périmètre de Cotre [5]. Il a pour but de développer un atelier de développement logiciel *open source* de façon à garantir une maintenance possible sur dix, vingt ou trente ans. La modélisation repose entre autres sur UML, AADL et l'extension Cotre, des automates, etc. Ce projet démontre que les démarches basées sur l'utilisation des ADL sont maintenant matures et peuvent être intégrées à un processus de développement pragmatique et industrialisé.

5.3. AEE et East-EEA

Même si l'on évoque beaucoup des domaines aéronautiques et spatiaux lorsque l'on parle de l'utilisation des ADL pour le développement de systèmes embarqués, l'industrie automobile n'est pas en reste.

Le projet français AEE avait pour but *de concevoir et de valider un processus rapide et sûr pour la définition de l'architecture système et le développement des logiciels associés, embarqués à bord des moyens de transport, notamment de l'automobile* [8, 4]. Il a en particulier définit un langage de description d'architectures véhicule nommé AIL-transport. L'intérêt de ce langage est de disposer d'un cadre sans rupture de la spécification des systèmes électroniques jusqu'à leur implantation.

Le projet AEE a été prolongé dans le cadre du programme européen ITEA par le projet East-EEA. Dans le cadre de ce projet, un nouvel ADL automobile a été défini basé sur AIL-transport, il s'agit de EAST-ADL [7].

Au début du projet East-EEA certaines capacités d'AADL ont semblé insuffisantes pour l'utiliser dans ce contexte, bien que le propos soit similaire. En particulier la modélisation de certains aspects importants pour l'automobile n'était pas prise en compte, par exemple la gestion des variantes ou certaines descriptions détaillées. Ces projets ont donc fait le choix de définir un ADL dédié à l'automobile et comportant nombre de mécanismes et éléments spécifiques. Cependant de futurs développements pourraient faire se rejoindre ces deux approches et définir une base commune.

6. Conclusion

Les langages de description d'architectures commencent à être introduits dans les processus industriels de développement de systèmes embarqués. Différents projets ont en effet montré qu'ils étaient une réponse possible aux besoin de l'industrie pour faire face à la complexité croissante et aux exigences nouvelles du marché.

L'intérêt suscité par ces ADL s'explique par leur capacité à supporter des méthodes et processus de développement innovants. Les quelques projets présentés portent tous d'abord sur les méthodes, les ADL sont pour eux un moyen et non une fin. Après une longue phase d'étude en laboratoire, et maintenant dans des projets de recherche pilotés par les industriels, on peut prévoir que des ADL fassent prochainement leur entrée dans les phases de production. AADL pourrait être un de ceux-là.

Références

[1] Site web aadl. `http://www.aadl.info`.
[2] Site web assert. `http://www.assert-online.net`.
[3] Site web cotre. `http://www.laas.fr/COTRE/frindex.html`.
[4] Site web east-eea. `http://www.east-eea.net`.
[5] Site web topcased. `http://www.topcased.org/`.
[6] Axlog Ingénierie. Aadl (architecture analysis & design language. `http://www.axlog.fr/R_d/aadl/`.
[7] EAST-EEA. Definition of language for automotive embedded electronic architecture. juin 2004.
[8] INRIA. East-eea : Architecture electronique embarquée. `http://www.inria.fr/valorisation/actions-nationales/aee.fr.html`.
[9] SAE. Architecture analysis & design language (aadl). (AS5506), novembre 2004.
[10] S. Vestal. Metah user's manual. 1998.

Thème 2

Model-Checking Temporel,
Vérification Probabiliste, Techniques de Tests

Elements of Model Checking

Stephan Merz
INRIA Lorraine & LORIA
Stephan.Merz@loria.fr

***Abstract*— We survey principles of model checking techniques for the automatic verification and debugging of reactive systems. These are formally represented as transition systems, and their properties are usually expressed in temporal logics. We survey the relationship between temporal logic and ω-automata, and introduce basic model checking algorithms for linear- and branching-time temporal logics. We discuss symbolic model checking and and reduction techniques. The paper ends with pointers to some more advanced topics, and serves as an annotated bibliography to some of the literature.**

I. Introduction

In just over 20 years, from the appearance of the first scientific articles on the subject at the beginning of the 1980s to the first decade of the 21st century, model checking has evolved from a somewhat esoteric research topic of logicians and theorists into a technology that is routinely applied to the analysis of hardware and, increasingly, software systems. During the same time, the more traditional approach of deductive verification has been much less successful as far as practical applications are concerned. The success of model checking is due to a mix of factors: it is basically a push-button technique: engineers can write (more or less abstract) models of their systems in (more or less) standard design languages, and they can ask questions such as: "will the system deadlock?", "will data arrive at their destination?", "does the protocol ensure cache coherence?". In the case of a negative answer, model checkers provide an explanation of the failure: they exhibit an execution of the model that leads to an error situation, and this output can be analyzed in very concrete terms. In many cases, the detected problem will be due to an error in modeling the system, and the model checking run will have to be repeated for some iterations. In other cases, the model checker may run out of memory or fail to produce an answer in a reasonable time; the model will then have to be simplified in order to make it amenable to model checking. Still, the technique has been found to be a valuable debugging aid on substantial examples, in relatively early phases of system design. Unlike testing, model checking is an *exhaustive* analysis technique; all border cases will be considered. Still, the limitations mentioned above require users to construct relatively coarse models of their systems in order to obtain a verdict at all, and bugs may well hide in the simplifications that were applied (often not very rigorously) in order to obtain the abstract model. Model checking is not a panacea and does not replace standard development processes and validation techniques such as code reviews.

Parts of this paper are based on [87].

As a scientific subject, model checking arose at the crossroads of logic, concurrency theory, automata theory, and algorithms. At this time, advances in model checking technology constitute a (or even the) major subject of conferences such as CAV, TACAS, CHARME, and others, and new ideas continue to make for lively discussions. Students considering to get into this field should have a solid background in computer science theory and logic, but should also be prepared to validate their ideas via concrete implementations and do extensive benchmarking to evaluate their applicability.

This paper intends to give an overview, at an elementary but precise level, of some of the most fundamental approaches and techniques to model checking. It is based on a (necessarily biased) selection of the large body of literature on the topic. Section II reviews transition systems, temporal logics, and automata-theoretic techniques that underly some approaches to model checking. Section III introduces basic model checking algorithms for linear-time and branching-time logics. Finally, Sect. IV collects some rather sketchy references to more advanced topics. Much more material can be found in other contributions to this volume and in the textbooks and survey papers [23], [24], [63], [85], [108] on the subject. This paper contains many references, and it is my hope that it will be useful as an annotated bibliography.

II. Systems and Properties

Reactive systems can be broadly classified as *distributed* systems whose subcomponents are spatially separated and *concurrent* systems that share resources such as processors and memories. Distributed systems communicate by *message passing*, whereas concurrent systems may use *shared variables*. Concurrent processes may share a common clock and execute in lock-step (*time-synchronous* systems, typical for hardware verification problems) or operate asynchronously, maybe sharing the processor. In the latter case, one will typically assume *fairness conditions* that ensure processes that could execute are eventually scheduled for execution. Despite this variety of concrete systems, they can formally be represented in a common framework of *transition systems*. Properties of transition systems are conveniently expressed in temporal logics.

A. Transition Systems

Definition 1: A *transition system* $\mathcal{T} = (S, I, \mathcal{A}, \delta)$ is given by a set S of *states*, a non-empty subset $I \subseteq S$ of *initial* states, a set $\mathcal{A}$ of *actions*, and a *transition relation* $\delta \subseteq S \times \mathcal{A} \times S$. It is convenient to assume that for every state $s \in S$ there exist $A \in \mathcal{A}$ and $t \in S$ such that $(s, A, t) \in \delta$.

An action $A \in \mathcal{A}$ is called *enabled* at state $s \in S$ iff $(s, A, t) \in \delta$ holds for some $t \in S$.

A *run* of $\mathcal{T}$ is an infinite sequence $\rho = s_0 s_1 \ldots$ of states $s_i \in S$ such that $s_0 \in I$ and for all $i \in \mathbb{N}$, $(s_i, A_i, s_{i+1}) \in \delta$ holds for some $A_i \in \mathcal{A}$.

A state $s \in S$ is *reachable* if $s_i = s$ for some run $\rho = s_0 s_1 \ldots$ of $\mathcal{T}$ and for some $i \in \mathbb{N}$. System $\mathcal{T}$ is called *finite-state* if the set of reachable states is finite.

A transition system specifies the allowed evolutions of the system: starting from some initial state, the system evolves by performing actions that take the system to a new state. Variations of this basic notion of transition systems abound in the literature. For example, actions are sometimes not explicitly identified. We assume the transition relation to be total in order to simplify some of the definitions below. One way to ensure totality is to require a special *stuttering action* that is always enabled and that does not change the state. More importantly, transition systems are often augmented by fairness conditions in order to exclude unfair runs (cf. section III-B). Transition systems are related to the concept of *Kripke structures* [69] that underly modal and temporal logics and where states are labelled by the set of propositions they satisfy.

Whereas transition systems are almost universally used as a formalism to describe the semantics of systems, model checkers accept *descriptions* of transition systems that are expressed in some modelling language rather than requiring the user to explicitly define a transition system. Typical description languages include (pseudo) programming languages such as PROMELA, but also process algebras or Petri nets, and the operational semantics of these languages is expressed in terms of transition systems. The distinction is relevant because the descriptions are typically exponentially more concise than the transition systems themselves, and a short description can nevertheless induce a large state space. For example, the state space of a multi-process program is the product of the state spaces of the individual processes. Certain optimizations, such as partial-order reduction methods (cf. section III-D), rely on the process structure of models that has been "flattened out" in the definition of transition systems above.

B. Invariant checking

A property is an *invariant* if it holds of all reachable states. Invariants express basic correctness requirements of systems, and they are the basis for the verification of more complex properties. Formally, assume given a denumerable set $\mathcal{V}$ of atomic propositions that can be evaluated to be true or false in any system state; we write $s(v) \in \{tt, ff\}$ to denote the value of proposition v at state s. An *assertion* P is a Boolean combination of propositions $v \in \mathcal{V}$, and we write $s \models P$ if s satisfies P. For example, if $P \equiv v_1 \wedge \neg v_2$ then $s \models P$ if and only if $s(v_1) = tt$ and $s(v_2) = ff$. Assertion P is an invariant of transition system $\mathcal{T}$ if $s \models P$ holds whenever s is a reachable state of $\mathcal{T}$.

Consider an assertion such as $\neg(own_1 \wedge own_2)$ expressing that two processes do not both hold a shared resource; this assertion is an invariant if the system guarantees that access to the resource is mutually exclusive. For another example, assume that a system can perform n possible actions whose enabling conditions (guards) are given as $g_1, \ldots, g_n$. The assertion $g_1 \vee \ldots \vee g_n$ holds if one of the system's actions is enabled; it is an invariant if the system never deadlocks.

```
Set visited = new Set();
Set todo = ts.getInitials();
while (!todo.isEmpty()) {
  State s = todo.someElement();
  todo.remove(s); visited.add(s);
  if (!s.satisfies(inv))
    report invariant violation and exit;
  foreach (s' in s.successors()) {
    if (!todo.contains(s') &&
        !visited.contains(s'))
      todo.add(s')
  }
}
report invariant satisfied
```

Fig. 1. Pseudo code for invariant checking.

For finite-state systems, invariants can be verified by systematically generating all reachable states and evaluating the assertion in each state. Figure 1 shows a pseudo-code representation of an algorithm that checks invariant `inv` for transition system `ts`. It is based on two sets `visited` and `todo` that hold the sets of states that have already been explored and that need to be checked (the union of these two sets contains the set of states that are so far known to be reachable). At the beginning of the verification, set `visited` is empty whereas `todo` contains the initial states of the transition system. As long as there are states to explore, the algorithm chooses some such state `s` and moves it to set `visited`. It reports an error if `inv` does not hold of `s`. Otherwise, all successor states of `s` that are encountered for the first time are added to `todo`. Because `ts` is assumed finite-state, the algorithm will eventually generate all reachable states and then terminate.

Despite its simplicity, the algorithm of Fig. 1 contains the kernel for the more elaborate model checking algorithms that we will discuss in Sect. III. Implementations will fix the organization of the set `todo` and the method `choose`, and thus determine the search strategy. In the absence of more specific information, `todo` can be organized as a queue or a stack, resulting respectively in breadth-first or depth-first search. A desirable additional feature of an implementation is to provide feedback to the user when the invariant does not hold by outputting a *counter-example*, a finite run leading to a state that violates the assertion. This is particularly easy for depth-first search: the stack can be organized in such a way that it contains the trace of states explored before the invariant violation. On the other hand, a breadth-first search guarantees that counter-examples of minimal length will be produced, which are usually easier to understand. Practical challenges when implementing an invariant checker are to be able to handle large sets of states and to provide an expressive modeling language while providing reasonable efficiency. Possible techniques include storing states on the disk in addition to main memory and reduction techniques,

such as those that will be mentioned in Sect. III-D. The Murφ system [34] is an example of a mature invariant checker.

C. Properties and Temporal Logic

Not all correctness properties of interest can be formulated as invariants. Below are some examples for typical correctness properties that need a richer language:

- Will every request (e.g., to access a shared resource) by some process eventually be honored?
- Are there runs such that, from some point onwards, some "desired" state is never reached or some action never executed? Can the system get into a livelock where some of its components never make progress although the system as a whole is not deadlocked?
- Is some initial system state of $\mathcal{T}$ reachable from every state? In other words, can the system be reset?

Such properties are the domain of temporal logic [39], [70], [82], [83], [102], a family of logics that have their roots in the study of temporal relationships in natural language, but that are now associated with system verification. Let us first consider temporal logic of linear time whose formulas express properties of runs of transition systems. Again, assume given a denumerable set $\mathcal{V}$ of atomic propositions.

Definition 2: Formulas of propositional temporal logic **PTL** of linear time are inductively defined as follows:

- Every atomic proposition $v \in \mathcal{V}$ is a formula.
- Boolean combinations of formulas are formulas.
- If φ and ψ are formulas then so are $\mathbf{X}\,\varphi$ ("*next* φ") and $\varphi\ \mathbf{U}\ \psi$ ("φ *until* ψ").

PTL formulas are interpreted over *behaviors*, which are ω-sequences of states. Again, we write $s(v)$ to denote the truth value of proposition $v \in \mathcal{V}$ at state s. For a behavior $\sigma = s_0 s_1 \ldots$, we let σ_i denote the state s_i and $\sigma|_i$ the suffix $s_i s_{i+1} \ldots$ of σ.

Definition 3: The relation $\sigma \models \varphi$ ("φ *holds of* σ") is inductively defined as follows:[1]

- $\sigma \models v$, for $v \in \mathcal{V}$, iff $\sigma_0(v) = tt$.
- The semantics of boolean combinations is defined as usual.
- $\sigma \models \mathbf{X}\,\varphi$ iff $\sigma|_1 \models \varphi$.
- $\sigma \models \varphi\ \mathbf{U}\ \psi$ iff for some $k \geq 0$, $\sigma|_k \models \psi$ and $\sigma|_j \models \varphi$ holds for all $0 \leq j < k$.

Other useful **PTL** formulas can be introduced as abbreviations: $\mathbf{F}\,\varphi$ ("*finally* φ", "*eventually* φ") is defined as $\mathbf{true}\ \mathbf{U}\ \varphi$; it asserts that φ holds of some suffix. The dual formula $\mathbf{G}\,\varphi \equiv \neg\,\mathbf{F}\,\neg\varphi$ ("*globally* φ", "*always* φ") requires φ to hold of all suffixes. The formula $\varphi\ \mathbf{W}\ \psi$ ("φ *waits for* ψ", "φ *unless* ψ") is defined as $(\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\,\varphi$ and requires φ to hold for as long as ψ does not hold; unlike $\varphi\ \mathbf{U}\ \psi$, it does not require ψ to become true eventually. Temporal operators can of course be nested; for example, $\mathbf{G}\,\mathbf{F}\,\varphi$ can be read as "φ holds infinitely often", and $\mathbf{F}\,\mathbf{G}\,\varphi$ as "φ holds almost always" (always from some point onward).

The following **PTL** formulas represent typical correctness assertions for a two-process resource manager. We assume rq_i and own_i to be atomic propositions true when process i has requested the resource or when it owns the resource.

$\mathbf{G}\,\neg(own_1 \wedge own_2)$: It is never the case that both processes own the resource. In general, properties of the form $\mathbf{G}\,p$, for non-temporal formulas p, express *invariants*.

$\mathbf{G}(rq_1 \Rightarrow \mathbf{F}\,own_1)$: Whenever process 1 has requested the resource, it will eventually obtain it. Formulas of this form are often called *response properties* [81].

$\mathbf{G}\,\mathbf{F}(rq_1 \wedge \neg(own_1 \vee own_2)) \Rightarrow \mathbf{G}\,\mathbf{F}\,own_1$: If it is infinitely often the case that process 1 has requested the resource when the resource is free, then process 1 infinitely often owns the resource. This formula expresses a (strong) fairness condition for process 1.

$\mathbf{G}(rq_1 \wedge rq_2 \Rightarrow (\neg own_2\ \mathbf{W}\ (own_2\ \mathbf{W}\ (\neg own_2\ \mathbf{W}\ own_1))))$: Whenever the two processes compete for the resource, process 2 will be granted the resource at most once before it is granted to process 1. This property, known as "1-bounded overtaking", is an example for a *precedence property*. It is easiest understood as asserting the existence of four, possibly empty or right-open, intervals that satisfy the respective conditions.

PTL formulas are true or false of single behaviors, but we are more interested in *system validity*: we say that formula φ holds of $\mathcal{T}$ (written $\mathcal{T} \models \varphi$) if φ is true of all runs of $\mathcal{T}$. In this sense, **PTL** formulas express *correctness properties* of a system. On the other hand, the existence of runs satisfying certain properties cannot be expressed in **PTL**. Such *possibility properties* are the domain of branching-time logics. A well-known representative of this family is the logic **CTL** (*computation tree logic* [21]).

Definition 4: Formulas of propositional **CTL** are inductively defined as follows:

- Every atomic proposition $v \in \mathcal{V}$ is a formula.
- Boolean combinations of formulas are formulas.
- If φ and ψ are formulas then $\mathbf{EX}\,\varphi$, $\mathbf{EG}\,\varphi$, and $\varphi\ \mathbf{EU}\ \psi$ are formulas.

CTL formulas are interpreted at the states of a transition system. A *path* in $\mathcal{T}$ is an ω-sequence $\sigma = s_0 s_1 \ldots$ of states related by δ; it is an s*-path* if $s = s_0$.

Definition 5: The relation $\mathcal{T}, s \models \varphi$ is inductively defined as follows:

- $\mathcal{T}, s \models v$ (for $v \in \mathcal{V}$) iff $s(v) = tt$.
- The semantics of boolean combinations is defined in the standard way.
- $\mathcal{T}, s \models \mathbf{EX}\,\varphi$ iff there exists an s-path $s_0 s_1 \ldots$ such that $\mathcal{T}, s_1 \models \varphi$.
- $\mathcal{T}, s \models \mathbf{EG}\,\varphi$ iff there is an s-path $s_0 s_1 \ldots$ such that $\mathcal{T}, s_i \models \varphi$ holds for all i.
- $\mathcal{T}, s \models \varphi\ \mathbf{EU}\ \psi$ iff there exist an s-path $s_0 s_1 \ldots$ and $k \geq 0$ such that $\mathcal{T}, s_k \models \psi$ and $\mathcal{T}, s_j \models \varphi$ holds for all $0 \leq j < k$.

Derived **CTL**-formulas include $\mathbf{EF}\,\varphi \equiv \mathbf{true}\ \mathbf{EU}\ \varphi$, $\mathbf{AX}\,\varphi \equiv \neg\,\mathbf{EX}\,\neg\varphi$, and $\mathbf{AG}\,\varphi \equiv \neg\,\mathbf{EF}\,\neg\varphi$. For example, the formula $\mathbf{AG}\,\neg(owns_1 \wedge owns_2)$ expresses mutual exclusion for the two-process resource manager, whereas $\mathbf{AG}(req_1 \Rightarrow \mathbf{EF}\,owns_1)$ asserts that whenever process 1 requests the resource, it *is possible* for it to eventually obtain the resource,

[1] Different authors use slightly different definitions of **PTL** semantics, which can lead to confusion.

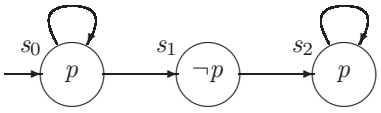


Fig. 2. A transition system $\mathcal{T}$ such that $\mathcal{T} \models \mathbf{F}\,\mathbf{G}\,p$ but $\mathcal{T} \not\models \mathbf{AF}\,\mathbf{AG}\,p$.

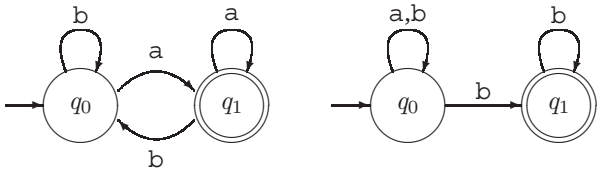


Fig. 3. Two Büchi automata.

although there may be executions that do not honor the request. The formula $\mathbf{AG}\,\mathbf{EF}\,\mathit{init}$ (for a suitable predicate init) asserts that the system is resettable.

System validity for **CTL**-formulas is defined by $\mathcal{T} \models \varphi$ if $\mathcal{T}, s \models \varphi$ holds for all initial states s of $\mathcal{T}$. The expressiveness of **PTL** and **CTL** can be compared by analyzing which properties of transition systems can be formulated. It turns out that neither logic subsumes the other one [76], [35], [37]: obviously, **PTL** cannot express possibility properties. Perhaps more surprisingly, it turns out that fairness properties cannot be stated in **CTL**. More specifically, there is no **CTL** formula that is system valid iff the **PTL** formula $\mathbf{F}\,\mathbf{G}\,\varphi$ is. In particular, the latter does not correspond to the **CTL** formula $\mathbf{AF}\,\mathbf{AG}\,\varphi$, as shown in Fig. 2: every run of the transition system $\mathcal{T}$ satisfies $\mathbf{F}\,\mathbf{G}\,p$ (either it stays in state s_0 forever or it ends in state s_2), but $\mathcal{T}, s_0 \not\models \mathbf{AF}\,\mathbf{AG}\,p$ (for the run that stays in state s_0 there is always the possibility to move to state s_1).

Extensions and variations.: The lack of expressiveness of **CTL** is due to the requirement that path quantifiers (**E**, **A**) and temporal operators (**X**, **G**, **U**) are paired together. The logic **CTL*** [35], [37] removes this requirement and (strictly) subsumes both **PTL** and **CTL**. For example, the **CTL*** formula $\mathbf{AFG}\,p$ is system valid iff the **PTL** formula $\mathbf{F}\,\mathbf{G}\,p$ is.

The *propositional μ-calculus* [68], also known as μ**TL**, allows properties to be defined as smallest or greatest fixed points, generalizing recursive characterizations of temporal operators such as

$$\mathbf{EG}\,\varphi \quad \equiv \quad \varphi \wedge \mathbf{EX}\,\mathbf{EG}\,\varphi$$

It strictly subsumes the logic **CTL***. For example, the formula $\nu X.\, \varphi \wedge \mathbf{AX}\,\mathbf{AX}\,X$ asserts that φ holds at every state with even distance from the current state.

Alternating-time temporal logic [3] refines the path quantifiers of branching time temporal logics by allowing references to different processes (or agents) of a reactive system. One can, for example, assert that the resource manager can ensure mutual exclusion between the clients, or that the manager and client 1 can cooperate to prevent client 2 to access the resource.

Past-time temporal connectives can be defined as "mirror images" of the connectives considered here. For example, $\mathbf{H}\,\varphi$ holds if φ has been true at all previous states of a behavior. It has been shown that for any formula φ with past-time operators there exists a "future" formula ψ such that $\mathcal{T} \models \varphi$ if and only if $\mathcal{T} \models \psi$ holds for any transition system $\mathcal{T}$, making past-time operators appear superfluous (see [48] for an extensive discussion). Nevertheless, past-time operators can make formulas often easier to read and sometimes exponentially more succinct [77].

Relativized connectives can be useful for the specification of real-time or probabilistic systems. For example $\mathbf{AF}_{\leq 5}\,\varphi$ can be read as asserting that φ must become true within 5 units of time [2], [58], and $\mathbf{EF}_{\geq .9}\,\varphi$ as saying that φ becomes true with at least 90% probability [57], [89].

D. ω-Automata

Whereas temporal logics are a convenient language for expressing correctness properties of transition systems, the quantification over all (infinite!) paths makes their semantics definition difficult to evaluate mechanically. The theory of finite automata turns out to be helpful in this respect, and close connections exist between temporal logics and automata on infinite objects (such as words or trees). In fact, many properties of finite automata are decidable. The theory of automata over infinite words and trees was initiated by Büchi [16], Muller [90], and Rabin [97]. We present some elementary results; for more comprehensive expositions see the excellent survey articles by Thomas [105], [106].

Definition 6: A *Büchi automaton* $\mathcal{B} = (Q, I, \delta, F)$ over an alphabet Σ is given by a finite set Q of *locations*[2], a non-empty set $I \subseteq Q$ of *initial locations*, a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and a set $F \subseteq Q$ of *accepting locations*.

A *run* of $\mathcal{B}$ over an ω-word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0 q_1 \ldots$ of locations $q_i \in Q$ such that $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$. The run ρ is *accepting* iff there exists some $q \in F$ such that $q_i = q$ holds for infinitely many $i \in \mathbb{N}$.

The *language* $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$ is the set of ω-words for which there exists some accepting run ρ of $\mathcal{B}$. A language $L \subseteq \Sigma^\omega$ is called *ω-regular* if $L = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$.

Büchi automata are presented just as ordinary (non-deterministic) finite automata over finite words [62]. The notion of "final locations", which obviously does not apply to ω-words, is replaced by the requirement that a run passes infinitely often through an accepting location. Figure 3 shows two Büchi automata of two locations with initial location q_0 and accepting location q_1. The language of the left-hand automaton is the set of ω-words over $\{\mathtt{a}, \mathtt{b}\}$ that contain infinitely many $\mathtt{a}$'s. The right-hand automaton accepts precisely those words that contain only finitely many $\mathtt{a}$'s.

Many properties of classical finite automata carry over to Büchi automata. For example, the emptiness problem is decidable.

Theorem 7: For a Büchi automaton $\mathcal{B}$, it is decidable in time linear in the size of $\mathcal{B}$ whether $\mathcal{L}(\mathcal{B}) = \emptyset$.

Proof: Because Q is finite, $\mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if there exist locations $q_0 \in I$, $q \in F$ and finite words $x \in \Sigma^*$ and

[2]We prefer to speak of *locations* rather than *states* to avoid confusion with the states of transition systems and temporal logic.

$y \in \Sigma^+$ such that $q_0 \stackrel{x}{\Rightarrow} q$ and $q \stackrel{y}{\Rightarrow} q$ (where $q \stackrel{w}{\Rightarrow} q'$ means that there is a path in $\mathcal{B}$ from location q to q' labelled with w). The existence of such paths can be decided in linear time using Tarjan's algorithm [104] to enumerate the strongly connected components of $\mathcal{B}$ reachable from locations in I, and checking whether some SCC contains some accepting location. ■

Observe that the construction used in the proof of theorem 7 implies that an ω-regular language is non-empty iff it contains some word of the form xy^ω where $x \in \Sigma^*$ and $y \in \Sigma^+$.

Unlike finite automata over finite words, deterministic Büchi automata are strictly weaker than non-deterministic ones. For example, there is no deterministic Büchi automaton that accepts the same language as the automaton shown on the right-hand side of Fig. 3. Intuitively, the reason is that unbounded non-determinism is required to "guess" when the last input a has been seen (a rigorous proof appears in e.g. [105]). It is therefore not obvious that the class of ω-regular languages is closed under complement: the familiar argument consisting of determinizing the automaton and then complement the set of accepting locations clearly fails. Still, the two automata of Fig. 3 accept complementary languages, and in general, Büchi [16] has shown that the complement of an ω-regular language is again ω-regular. A succession of papers has improved on Büchi's original, non-constructive proof, culminating in a construction due to Safra [98] of essentially optimal complexity. More recently, Kupferman and Vardi [74], [73] have presented simpler proofs of the following result, that apply uniformly to several classes of ω-automata.

Proposition 8: For a Büchi automaton $\mathcal{B}$ with n locations over alphabet Σ there is a Büchi automaton $\overline{\mathcal{B}}$ with $2^{O(n \log n)}$ locations such that $\mathcal{L}(\overline{\mathcal{B}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.

Other types of ω-automata have also been considered. *Generalized Büchi automata* define the acceptance condition by a (finite) set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets of locations [110]. A run is accepting if some location from each F_i is visited infinitely often. Using a counter modulo n, it is not difficult to simulate a generalized Büchi automaton by a standard one. The algorithm for checking nonemptiness can be adapted by searching some strongly connected component that contains some location from every F_i. *Muller automata* also specify the acceptance condition as a set $\mathcal{F}$ of set of locations; a run is accepting if the set of locations that appears infinitely often is an element of $\mathcal{F}$. Rabin and Streett automata define acceptance conditions in terms of pairs of sets of locations, such as requiring that if locations in a set $R \subseteq Q$ are visited infinitely often then there are also infinitely many visits to locations in another set $G \subseteq Q$. Streett automata can be exponentially more succinct than Büchi automata, and deterministic Rabin and Streett automata are at the heart of Safra's complementation proof. It is also possible to place acceptance conditions on the transitions rather than the locations [4].

Alternating automata [91] present a more radical departure from the format of Büchi automata and have attracted much interest in recent years. The basic idea is to allow the automaton to make a transition from one location to several successor locations that are simultaneously active. One way to define such a relation is to let $\delta(q, a)$ be a positive Boolean formula with the locations as atomic propositions. For example,

$$\delta(q_1, a) \quad = \quad (q_2 \wedge q_3) \vee q_4$$

specifies that whenever location q_1 is active and input symbol $a \in \Sigma$ is read, the automaton moves to locations q_2 and q_3 in parallel, or to location q_4. Runs of alternating automata are no longer infinite sequences, but rather infinite trees or dags of locations. Although they also define the class of ω-regular languages, alternating automata can be exponentially more succinct than Büchi automata, due to their inherent parallelism. Complementation of alternating automata can be much simpler (and often of linear complexity), depending on the precise type of acceptance condition.

E. Temporal Logic and Automata

Recall that **PTL** is interpreted over behaviors, which are ω-sequences of states, and can be interpreted as ω-words over the alphabet $2^{\mathcal{V}}$, identifying a system state s and the set of atomic propositions v for which $s(v) = tt$. From this perspective, **PTL** formulas and ω-automata are two different formalisms to describe ω-words, and it is interesting to compare their expressiveness. For example, the Büchi automata of Fig. 3 can be identified with the **PTL** formulas $\mathbf{G}\,\mathbf{F}\,\mathtt{a}$ and $\mathbf{F}\,\mathbf{G}\,\mathtt{b}$.

Translations from **PTL** to equivalent Büchi automata will allow us to apply algorithms designed for automata to the realm of logic, and in particular to **PTL** model checking. We now outline a construction that produces a generalized Büchi automaton $\mathcal{B}_\varphi$ from a given **PTL** formula φ such that $\mathcal{B}_\varphi$ accepts precisely those runs over which φ holds. In view of the high complexity of complementation (cf. Prop. 8), the construction is not defined by induction on the structure of φ but is based on a "global" construction that considers all subformulas of φ simultaneously. The *Fischer-Ladner closure* $\mathcal{C}(\varphi)$ of formula φ is the set of subformulas of φ and their complements, identifying $\neg\neg\psi$ and ψ. The locations of $\mathcal{B}_\varphi$ are subsets of $\mathcal{C}(\varphi)$, with the intuition that an accepting run of $\mathcal{B}_\varphi$ from location q satisfies the formulas in q. More precisely, the locations q of $\mathcal{B}_\varphi$ are all subsets of $\mathcal{C}(\varphi)$ that satisfy the following *healthiness conditions*:

- For all $\psi \in \mathcal{C}(\varphi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- If $\psi_1 \vee \psi_2 \in \mathcal{C}(\varphi)$ then $\psi_1 \vee \psi_2 \in q$ iff $\psi_1 \in q$ or $\psi_2 \in q$.
- Conditions for other boolean combinations are similar.
- If $\psi_1\ \mathbf{U}\ \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\psi_1\ \mathbf{U}\ \psi_2 \in \mathcal{C}(\varphi) \setminus q$, then $\psi_2 \notin q$.

The initial locations of $\mathcal{B}_\varphi$ are those locations containing φ. The transition relation δ of $\mathcal{B}_\varphi$ has $(q, s, q') \in \delta$ iff all of the following conditions hold:

- $s(v) = tt$ iff $v \in q$ for $v \in \mathcal{V}$: the state s must immediately satisfy precisely those atomic propositions "promised" by q.
- If $\mathbf{X}\,\psi \in \mathcal{C}(\varphi)$, then $\psi \in q'$ if and only if $\mathbf{X}\,\psi \in q$.
- If $\psi_1\ \mathbf{U}\ \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1\ \mathbf{U}\ \psi_2 \in q'$.
- If $\psi_1\ \mathbf{U}\ \psi_2 \in \mathcal{C}(\varphi) \setminus q$ and $\psi_1 \in q$ then $\psi_1\ \mathbf{U}\ \psi_2 \notin q'$.

The healthiness and next-state conditions are justified by propositional consistency and by the "recursion law"

$$\psi_1\ \mathbf{U}\ \psi_2 \quad \equiv \quad \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1\ \mathbf{U}\ \psi_2))$$

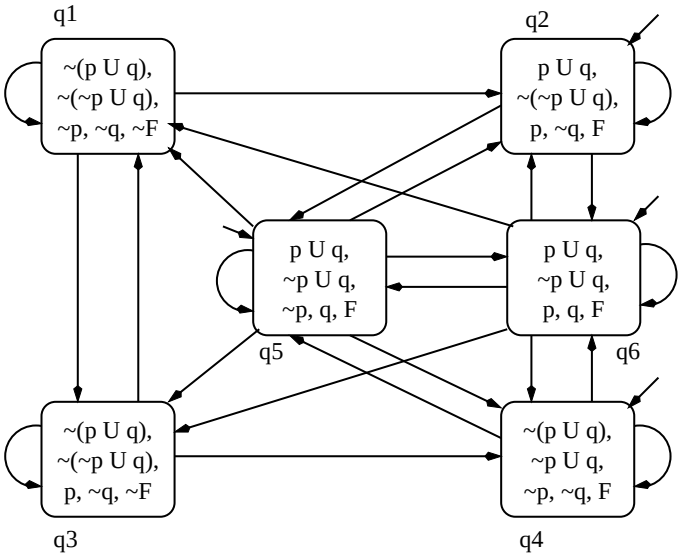


Fig. 4. Büchi automaton for $F \equiv (p \mathbf{U} q) \vee (\neg p \mathbf{U} q)$.

In particular, they ensure that whenever some location contains $\psi_1 \mathbf{U} \psi_2$, subsequent locations contain ψ_1 for as long as they do not contain ψ_2.

The healthiness and next-state conditions ensure that "locally" the semantic conditions of **PTL** operators are satisfied. However, it is still possible to "unroll" a formula $\psi_1 \mathbf{U} \psi_2$ indefinitely such that ψ_1 is always satisfied without ever satisfying ψ_2. The acceptance condition of $\mathcal{B}_\varphi$ will be defined just to rule out such behaviors. Let $\psi_1^1 \mathbf{U} \psi_2^1, \ldots, \psi_1^k \mathbf{U} \psi_2^k$ be all **U**-formulas in $\mathcal{C}(\varphi)$. Then $\mathcal{B}_\varphi$ has the acceptance condition $\mathcal{F} = \{F_1, \ldots, F_k\}$ where F_i is the set of locations that do not contain $\psi_1^i \mathbf{U} \psi_2^i$ or that contain ψ_2^i. As an example, Fig. 4 shows the automaton $\mathcal{B}_F$ obtained for the formula $F \equiv (p \mathbf{U} q) \vee (\neg p \mathbf{U} q)$. For clarity, we have omitted the edge labels, which are simply the set of atomic propositions contained in the source location. The acceptance sets corresponding to the subformulas $p \mathbf{U} q$ and $\neg p \mathbf{U} q$ are $\{q_1, q_3, q_4, q_5, q_6\}$ and $\{q_1, q_2, q_3, q_5, q_6\}$. For example, they ensure that no accepting run remains forever in location q_2.

This construction, which is very similar to a tableau construction [112], implies the existence of a Büchi automaton that accepts precisely the models of any given **PTL** formula. The following proposition is due to [78], [110].

Proposition 9: For every **PTL** formula φ of length n there exists a Büchi automaton $\mathcal{B}_\varphi$ with $2^{O(n)}$ locations that accepts precisely the behaviors of which φ holds.

Combining proposition 9 and theorem 7, it follows that the satisfiability problem for **PTL** is solvable in exponential time by checking whether $\mathcal{L}(\mathcal{B}_\varphi) = \emptyset$; in fact, Sistla and Clarke [101] have shown that the **PTL** satisfiability problem is PSPACE-complete. Note that the above construction invariably produces a Büchi automaton $\mathcal{B}_\varphi$ whose size is exponential in the length of the formula φ. Constructions that try to avoid this exponential blow-up [52], [32], [49] are at the basis of actual implementations. Alternatively, one can construct a weak alternating automaton corresponding to φ in linear time and perform an emptiness test (with exponential worst-case complexity) for that automaton, as suggested in [56].

In particular, ω-regular languages are at least as expressive as **PTL** formulas for the description of sets of behaviors. Kamp [67] showed that this inclusion is strict by proving that the expressiveness of **PTL** coincides with that of monadic first-order logic of linear orders with one free variable (formulas built from $=$, $<$, and unary predicates $P_v(x)$, for $v \in \mathcal{V}$, interpreted over the natural numbers), see also [48]. This fragment of first-order logic is known to define the set of *star-free* ω-regular languages, a result due to McNaughton and Papert [86], [106]. For example, the set of behaviors such that proposition p is true at the even positions (and may be true or false elsewhere) is ω-regular (exercise!) but not **PTL**-definable [112]. **PTL** can be augmented to attain the expressiveness of ω-regular languages (which Büchi showed to coincide with the monadic *second-order* theory of linear orders) by so-called "automaton operators" [112], by fixed-point formulas [102] or by quantification over atomic propositions. Unfortunately, the satisfiability problem for some of the resulting logics is of non-elementary complexity; moreover, few applications seem to require that level of expressiveness.

Automata for other temporal logics.: Automata-theoretic characterizations of branching-time logics [71] are based on tree automata [105], [106], which again define a notion of regular tree languages. Alternating automata allow for a rather uniform presentation of decision procedures for linear-time, branching-time, and alternating-time temporal logics [92], [109], [74], based on different restrictions on the automaton format. An essentially equivalent approach that does not mention automata can be formulated in terms of logical games [103]. In particular, winning strategies replace the traditional presentation of counter-examples; this can give better feedback to the user who can then explore different scenarios that violate a property.

III. Algorithms for Model Checking

Given a transition system $\mathcal{T}$ and a formula φ, the model checking problem calls for deciding whether $\mathcal{T} \models \varphi$ holds or not. In the negative case, we would also like the model checker to provide an explanation, in the form of a counterexample. Thus, it is an extension of the problem of invariant checking considered in Sect. II-B to arbitrary temporal logic properties.

In accordance with the two parameters of the model checking problem ($\mathcal{T}$ and φ), there are two basic strategies when designing a model checking algorithm: "global" algorithms recurse on the structure of φ and evaluate each of its subformulas over all states of $\mathcal{T}$. "Local" algorithms, in contrast, attempt to limit their search to relevant parts of the of the state space, but check all subformulas of φ in the process. The choice between global and local model checking algorithms does not affect the worst-case complexity of model checking, but the average behavior on practical examples can differ greatly. Traditionally, local algorithms dominate for **PTL** model checking, while model checkers for **CTL** and other branching-time logics have used global algorithms. Whatever algorithm is used, success in practice hinges on efficient implementations and clever optimizations.

```
dfs(boolean search_cycle) {
  p = top(stack);
  foreach (q in successors(p)) {
    if (search_cycle and (q == seed))
      report acceptance cycle and exit;
    if ((q, search_cycle) not in visited) {
      push q onto stack;
      enter (q, search_cycle) into visited;
      dfs(search_cycle);
      if (!search_cycle && (q is accepting)) {
        seed = q; dfs(true);
  } } }
  pop(stack);
}
// initialization
stack = emptystack(); visited = emptyset();
seed = nil;
foreach initial pair p {
  push p onto stack;
  enter (p, false) into visited;
  dfs(false)
}
```

Fig. 5. On-the-fly **PTL** model checking algorithm.

A. *Local* **PTL** *Model Checking*

The model checking problem for **PTL** can be restated as follows: given $\mathcal{T}$ and φ, does there exist a run of $\mathcal{T}$ that does not satisfy φ? This is a refinement of the satisfiability problem considered in section II-E: instead of asking whether $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$, we now ask whether the language defined by the product of $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$ is empty or not. Observe that $\mathcal{T}$ can be considered as a Büchi automaton without acceptance conditions, and that the product $\mathcal{T} \times \mathcal{B}_{\neg\varphi}$ therefore yields another Büchi automaton. The basic algorithmic idea comes from (the proof of) Thm. 7: $\mathcal{T}$ has a run violating φ if and only if the product has an accepting location that is reachable from some initial state and (non-trivially) from itself.

Formally, assume given a finite transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta_{\mathcal{T}})$ and a Büchi automaton $\mathcal{B}_{\neg\varphi} = (Q, J, \delta_{\mathcal{B}}, F)$ that accepts precisely those behaviors that do not satisfy φ. The model checking algorithm operates on pairs (s, q) of system states and automaton locations. A pair (s_0, q_0) is *initial* if $s_0 \in I$ and $q_0 \in J$ are initial for $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$, respectively. A pair (s', q') is a *successor* of (s, q) if both $(s, A, s') \in \delta_{\mathcal{T}}$ (for some $A \in \mathcal{A}$) and $(q, s(\mathcal{V}), q') \in \delta_{\mathcal{B}}$ hold: $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$ make joint transitions, the input for $\mathcal{B}_{\neg\varphi}$ being determined by the values of the atomic propositions at the current system state. A pair (s, q) is *accepting* if $q \in F$ is an accepting automaton location.

The model checking algorithm shown in Fig. 5 is based on this setup. It is due to Courcoubetis et al [29] and can be understood as an extension of the algorithm for invariant checking of Fig. 1, for depth-first search. Notably, the algorithm interleaves the construction of (the relevant parts of) the product automaton and the search for acceptance cycles; such algorithms are called "on-the-fly" and avoid a construction of the full product automaton, which may well be too large to store in memory. The algorithm maintains a stack of pairs whose successors need to be explored and a set of pairs that have already been visited. Starting from the initial pairs, the procedure `dfs` generates reachable pairs until some accepting pair is found. At this point, the search switches to cycle search mode (indicated by the boolean parameter `search_cycle`) and tries to find a path that leads back to the accepting pair. Pairs that have already been encountered in the current search mode are not explored any further. Courcoubetis et al. [29] have shown that the algorithm will find some acceptance cycle if one exists, although it is not guaranteed to find all cycles (even if the search were continued instead of exiting). When an acceptance cycle is found, the sequence of system states contained in the stack represents a run of $\mathcal{T}$ that violates formula φ and can be displayed to the user as a counter-example.

For large models, storing the set of visited pairs may become a problem. If one is willing to trade complete coverage for the ability to analyze systems that would otherwise be unmanageable, one can instead maintain a set of *hash codes* of visited pairs, possibly using several hashing functions [60].

The model checking algorithm of Fig. 5 has time complexity linear in the product of the sizes of $\mathcal{T}$ and of $\mathcal{B}_{\neg\varphi}$; by proposition 9 the latter can be exponential in the size of φ. However, correctness assertions are often rather short, and as we mentioned in section II-A, the size of $\mathcal{T}$ can be exponential in the size of the description input to the model checker. Therefore, in practice the size of the transition system is the limiting factor. Given current technology, the analysis of systems on the order of 10^6–10^7 reachable states is feasible. Beyond this size, further optimization and reduction techniques become necessary; these will be discussed briefly in Sect. III-D.

Instead of the "double DFS search" algorithm of Fig. 5, **PTL** model checking algorithms can also be based on variants of Tarjan's algorithm. Schwoon and Esparza [50], [100] discuss tradeoffs and pitfalls for practical implementations.

B. *Global* **CTL** *Model Checking*

Let us now consider global model checking algorithms for the logic **CTL**. By $[\![\psi]\!]_{\mathcal{T}}$ (for a **CTL** formula ψ) we denote the set of states s of $\mathcal{T}$ such that $\mathcal{T}, s \models \psi$. The model checking problem can then be rephrased as deciding whether $I \subseteq [\![\varphi]\!]_{\mathcal{T}}$ holds. The satisfaction sets $[\![\psi]\!]_{\mathcal{T}}$ can be computed by induction on the structure of ψ, as follows:

$$
\begin{aligned}
[\![v]\!]_{\mathcal{T}} &= \{s : s(v) = tt\} \quad (\text{for } v \in \mathcal{V}) \\
[\![\neg\psi]\!]_{\mathcal{T}} &= S \setminus [\![\psi]\!]_{\mathcal{T}} \\
[\![\psi_1 \vee \psi_2]\!]_{\mathcal{T}} &= [\![\psi_1]\!]_{\mathcal{T}} \cup [\![\psi_2]\!]_{\mathcal{T}} \\
[\![\mathbf{EX}\,\psi]\!]_{\mathcal{T}} &= \delta^{-1}([\![\psi]\!]_{\mathcal{T}}) \\
&= \{s : \exists A, t : (s, A, t) \in \delta \text{ and } t \in [\![\psi]\!]_{\mathcal{T}}\} \\
[\![\mathbf{EG}\,\psi]\!]_{\mathcal{T}} &= \mathit{gfp}(\lambda X.[\![\psi]\!]_{\mathcal{T}} \cap \delta^{-1}(X)) \\
[\![\psi_1\ \mathbf{EU}\ \psi_2]\!]_{\mathcal{T}} &= \mathit{lfp}(\lambda X.[\![\psi_2]\!]_{\mathcal{T}} \cup ([\![\psi_1]\!]_{\mathcal{T}} \cap \delta^{-1}(X)))
\end{aligned}
$$

where $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$, for a function $f : 2^S \to 2^S$, denote the least and greatest fixed points of f. (These fixed points exist and can be computed effectively because S is finite.) The clauses for the **EG** and **EU** connectives are justified from the

recursive characterizations

$$\begin{aligned} \mathbf{EG}\,\psi &\equiv \psi \wedge \mathbf{EX}\,\mathbf{EG}\,\psi \\ \psi_1\ \mathbf{EU}\ \psi_2 &\equiv \psi_2 \vee (\psi_1 \wedge \mathbf{EX}(\psi_1\ \mathbf{EU}\ \psi_2)) \end{aligned}$$

The clause for **EU** calls for the computation of a least fixed point. Intuitively, this is because ψ_2 has to become true eventually, and thus the unfolding of the fixed point must eventually terminate. On the other hand, the greatest fixed point is required in the computation of $[\![\mathbf{EG}\,\psi]\!]$ because ψ has to hold arbitrarily far down the path. It is easy to see that the least fixed point of the function corresponding to $\mathbf{EG}\,\psi$ is the empty set, whereas the greatest fixed point in the case of **EU** computes $[\![\psi_1\ \mathbf{EW}\ \psi_2]\!]$.

For an implementation, we need to be able to efficiently calculate the *inverse image* function δ^{-1}. Sets $[\![\psi]\!]_{\mathcal{T}}$ that have already been computed can be memorized in order to avoid recomputation of common subformulas. In order to assess the complexity of the algorithm, first note that computation of the fixed points is at most cubic in $|S|$ (if the computation has not stabilized, at least one state is added to or removed from the current approximation per iteration, and every iteration may need to search the entire set of transitions, which may be quadratic in $|S|$). Second, there are as many recursive calls as φ has subformulas, so the overall complexity is linear in the length of φ and cubic in $|S|$.

Clarke, Emerson, and Sistla [25] have proposed a less naive algorithm whose complexity is linear in the product of the sizes of the formula and the model. For formulas $\psi_1\ \mathbf{EU}\ \psi_2$, the idea is to apply backward breadth-first search. For $\mathbf{EG}\,\psi$, first the model is restricted to states satisfying ψ (which have already been computed recursively), and the strongly connected components of this restricted graph are enumerated. The set $[\![\mathbf{EG}\,\psi]\!]_{\mathcal{T}}$ consists of all states of the restricted model from which some SCC can be reached; these states are again found using breadth-first search.

Because fairness assumptions can not be formulated in **CTL**, they must be specified as part of the model, and the model checking algorithm needs to be adapted accordingly. For example, the SMV model checker [85] allows to specify fairness constraints via **CTL** formulas. We define fair variants $\mathbf{EG}_f$ and $\mathbf{EU}_f$ of the **CTL** operators whose semantics is as in definition 5, except that quantifiers are restricted to fair paths, i.e., paths that contain infinitely many states satisfying the constraints. Let us call a state s *fair* iff there is some fair s-path; this is the case iff $\mathcal{T}, s \models \mathbf{EG}_f\,\mathbf{true}$ holds. It is easy to see that $\psi_1\ \mathbf{EU}_f\ \psi_2$ is equivalent to $\psi_1\ \mathbf{EU}\ (\psi_2 \wedge \mathbf{EG}_f\,\mathbf{true})$, hence we need only define an algorithm to compute $[\![\mathbf{EG}_f\,\psi]\!]_{\mathcal{T}}$. The algorithm of Clarke, Emerson, and Sistla can be modified by restricting to those SCCs that for each fairness constraint ζ_i contain some state satisfying ζ_i. The complexity of fair **CTL** model checking is thus still linear in the sizes of the formula and the model. For more information on different kinds of fairness constraints and their associated model checking algorithms see [36], [38], [72].

A global model checking algorithm for the branching-time fixed point logic μ**TL** can be defined along the same lines. The complexity is then of the order $|\varphi| \cdot |S|^{\,qd(\varphi)}$ where $qd(\varphi)$ denotes the nesting depth of the fixed point operators in the formula φ. However, Emerson and Lei [38] observed that the computation of fixed points can be optimized for blocks of fixed point operators of the same type, resulting in a complexity of order $|\varphi| \cdot |S|^{\,ad(\varphi)}$ where $ad(\varphi)$ is the alternation depth of fixed point operators of different type in φ. In particular, the complexity of model checking *alternation-free* μ**TL** is the same as for **CTL** [36], [27].

C. Symbolic model checking

The ability to analyze systems of relevant size using model checking requires efficient data structures to represent objects such as transition systems and sets of system states. Any finite-state system can be encoded using a set $\{b_1, \ldots, b_n\}$ of binary variables, just as ordinary data types of programming languages are represented in binary form on a digital computer. Sets of states, for example the set of initial states, can then be represented as propositional formulas over $\{b_1, \ldots, b_n\}$, and sets of pairs of states, such as the pairs (s, t) related by δ (for some action) can be represented as propositional formulas over $\{b_1, \ldots, b_n, b'_1, \ldots, b'_n\}$ where the unprimed variables represent the pre-state s and the primed variables represent the post-state t. The size of the representing formula depends on the structure of the represented set rather than on its size: for example, the empty set and the set of all states are represented by **false** and **true**, both of size 1. For this reason, such representations are often called *symbolic*, and model checking algorithms that work on symbolic representations are called *symbolic model checking* techniques [17], [85].

Binary decision diagrams [13], [15] (more precisely, reduced ordered BDDs) are a data structure for the symbolic representation of sets that have become very popular for model checking because they offer the following features:

- Every boolean function has a unique, canonical BDD representation. If sharing of BDD nodes is enforced, equality of two functions can be decided in constant time by checking for pointer equality.
- Boolean operations such as negation, conjunction, implication etc. can be implemented with complexity proportional to the product of the inputs.
- Projection (quantification over one or several boolean variables) is easily implemented; its complexity is exponential in the worst case but tends to be well behaved in practice.

BDDs can be understood as compact representations of ordered decision trees. For example, Fig. 6 shows a decision tree for the formula

$$(x_1 \wedge y_1) \vee ((x_1 \vee y_1) \wedge (x_0 \wedge y_0))$$

which is the characteristic function for the carry bit produced by an addition of the two-bit numbers x_1x_0 and y_1y_0. To find the result for a given input, follow the path labelled with the bit values for each of the inputs. The label of the leaf indicates the value of the function. The tree is ordered because the variables appear in the same order along every branch.

The decision tree of Fig. 6 contains many redundancies. For example, the values of y_0 and y_1 are irrelevant if x_0 and x_1 are

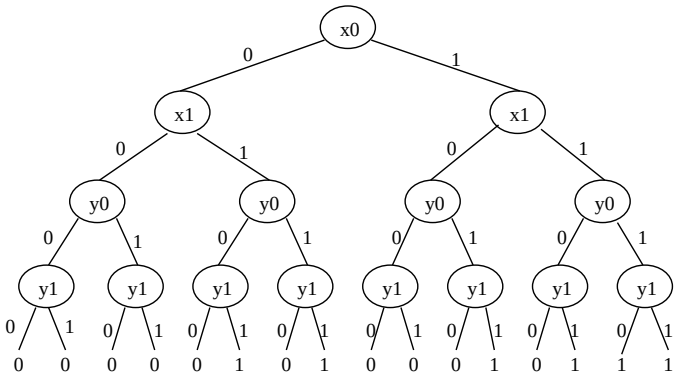


Fig. 6. Ordered decision tree for 2-bit carry.

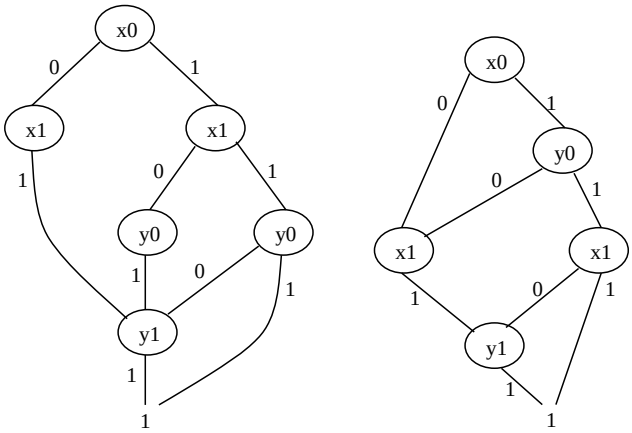


Fig. 7. BDDs for carry from 2-bit adder.

both 0. Similarly, y_0 is irrelevant in case x_0 is 0 and x_1 is 1. The redundancies can be removed by combining isomorphic subtrees (producing a directed acyclic graph from the tree) and eliminating nodes with identical subtrees. In our example, we obtain the BDD shown on the left-hand side of Fig. 7, where the leaf labelled 0 and all edges leading into it have been deleted for clarity. In an actual implementation, all BDD nodes that have been allocated are kept in a hash table indexed by the top variable and the two sub-BDDs, in order to avoid identical BDDs to be created twice. This ensures that two BDDs are functionally equivalent if and only if they are identical.

For a fixed variable ordering the BDD representing any given propositional formula is uniquely determined (and equivalent formulas are represented by the same BDD), but BDD sizes can vary greatly for different variable orderings. For example, the right-hand side of Fig. 7 shows a BDD for the same formula as before, but with the variable ordering x_0, y_0, x_1, y_1. When considering the carry for n-bit addition, the BDD sizes for the variable ordering $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}$ grow exponentially with n, whereas they grow only linearly for the ordering $x_0, y_0, \ldots, x_{n-1}, y_{n-1}$. It is usually a good heuristic to group "dependent" variables closely together [47], [41]. In general, however, the problem of finding an optimal variable ordering is NP-hard [14], and existing BDD libraries offer automatic reordering strategies based on steepest-ascent heuristics [46], [7]. There are also functions (such as multiplication) for which no variable ordering can avoid exponential growth. This is also a problem when representing queues, frequently necessary for the analysis of communication protocols, and special-purpose data structures have been suggested [10], [53].

Given two BDDs f and g (w.r.t. some fixed variable ordering) the BDD that corresponds to Boolean combinations such as $f \wedge g$, $f \vee g$ etc. can be constructed as follows:

- If f and g are both terminal BDDs (0 or 1), return the terminal BDD for the result of applying the operation.
- Otherwise, let v be the smaller of the variables at the root of f and g. Recursively apply the operation to the sub-BDDs that correspond to v being 0 and 1 (often called the "co-factors" of f and g for variable v). The results l and r correspond to the left- and right-hand branches of the result BDD. If $l = r$, return l, otherwise return a BDD with top variable v and children l and r.

When recursive calls to this "apply" function are memorized in a hash table, the number of subproblems to be solved is at most the number of pairs of nodes in f and g. Assuming perfect hashing, the complexity is therefore linear in the product of the sizes of f and g.

Observing that existential quantification over propositional variables can be computed as

$$(\exists v : f) \quad \equiv \quad f|_{v=0} \vee f|_{v=1}$$

the computation of a BDD corresponding to the quantified formula can be reduced to calculating co-factors and disjunction, and in fact quantification over a set of variables can be performed in a single pass over the BDD.

Symbolic **CTL** *model checking.:* The naive **CTL** model checking algorithm of section III-B is straightforward to implement based on a BDD representation of the transition system $\mathcal{T}$. It computes BDDs for the sets $[\![\psi]\!]_{\mathcal{T}}$; in particular, the inverse image $\delta^{-1}(X)$ of a set X that is represented as a BDD is computed as the BDD

$$\exists b'_1, \ldots, b'_n : \delta \wedge X'$$

where X' is a copy of X in which all variables have been primed, and $b'_1, \ldots, b'_n$ are all the primed variables. Naive computation of fixed points is also very simple using BDDs because equality of BDDs can be decided in constant time.

It is interesting to compare the complexity of this BDD-based algorithm with that of explicit-state **CTL** model checking: Because the representation of the transition relation using BDDs can be exponentially more succinct than an explicit enumeration, the symbolic algorithm has exponential worst-case complexity in terms of the BDD sizes for the transition relation. First, the number of iterations required for the calculation of the fixed points may be exponential in the number of the input variables, and secondly, the computation of the inverse image may produce BDDs exponential in the size of their inputs. In practice, however, the number of iterations required for stabilization is often quite small, and the inverse image operation is well-behaved. This holds especially for hardware verification problems of "regular" structure and with short data paths. (A precise definition of "regular" is, however, very

difficult.) For this class of problems, symbolic model checking has been successfully applied to the analysis of systems with 10^{100} states and more [26]. The main problem is then to find a variable ordering that yields a small representation of the transition system.

Symbolic model checking for other logics.: The approach used for symbolic **CTL** model checking extends basically unchanged for propositional μ**TL**. An extension for the richer *relational μ-calculus* [93] has been described by Burch et al. [17] and implemented in the model checker μcke [9].

Symbolic model checking for **PTL** has been considered in [20], [99]. The basic idea is to represent each formula in $\mathcal{C}(\varphi)$ by a boolean variable and to define the transition relation and acceptance condition of $\mathcal{B}_{\neg\varphi}$ in terms of these variables rather than constructing the automaton explicitly.

Bounded model checking.: Although symbolic model checking has traditionally been associated with BDDs, other representations of boolean functions are possible. *Bounded model checking* has become popular [8]. It relies on the observation that state sequences of fixed length, say k, can be represented using k copies of the variables used to represent a single state. The set of fixed-length sequences that represent terminating or looping runs of a given finite-state transition system $\mathcal{T}$ can therefore be encoded by formulas of (non-temporal) propositional logic, as well as the semantics of **PTL** formulas φ over such sequences. For any given length k, the existence of a state sequence of length k that represents a run of $\mathcal{T}$ satisfying φ can thus be reduced to the satisfiability of a certain propositional formula, which can be decided using Boolean satisfiability solvers. On the other hand, the *small model property* of **PTL** (which follows from the tableau-based decision procedure discussed in Sect. II-E) implies that there is a run of $\mathcal{T}$ satisfying φ if and only if there is some such run that can be represented by a sequence of length at most $|S| \cdot 2^{|\varphi|}$. A model checking algorithm is therefore obtained by enumerating all finite executions up to this bound.

D. Reduction techniques

Whereas symbolic model checking derives its power from efficient data structures for the representation and manipulation of large sets of sufficiently regular structure, algorithms based on explicit state enumeration can be improved if only a fraction of the reachable state space (more precisely, of the reachable pairs) has to be explored. Such techniques are known as "reductions", and the best-known instances are partial-order and symmetry reductions.

Partial-order reductions apply in the case of asynchronous systems that are composed of concurrent processes with relatively little interaction. The full transition system has as its runs all possible interleavings of the actions of the individual processes. For many properties, however, the relative order of concurrent actions is irrelevant, and it suffices to consider only a few sequentializations. More sophisticated models than simple interleaving-based representations have been considered in concurrency theory. In particular, *Mazurkiewicz traces* model runs as partial orders of events. Reduction techniques that take advantage of the commutativity of actions are therefore often called *partial-order reductions*, although the analogy to Mazurkiewicz traces is really quite superficial.

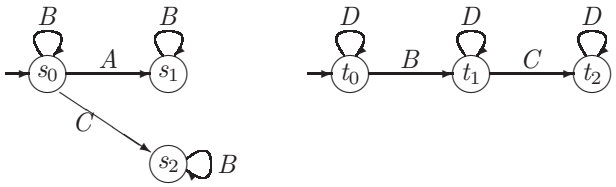


Fig. 8. Transition systems for two processes.

The main problem in the design of a practical algorithm is to detect when two actions commute, given only the "local" knowledge available at a given system state. For example, consider the transition systems for two processes represented in Fig. 8. The left-hand process has a choice between executing actions A and C, whereas the right-hand process must perform action B before action C. Assuming that processes synchronize on common actions, action C is disabled at the global state (s_0, t_0), whereas A, B, and D could be performed. Moreover, all these actions commute at state (s_0, t_0). In particular, A and B can be executed in either order, resulting in the global state (s_1, t_1). However, it would be an error to conclude that only the successors of state (s_0, t_0) with respect to action A need be considered, because action C can then never be taken. The lesson is that actions that are currently disabled must nevertheless be taken into account when constructing a reduced state space.

There is also a danger of prematurely stopping the state exploration because actions are delayed forever along a loop. For an extreme example, consider again the transition systems of Fig. 8 at the global state (s_0, t_0). The local action D of the right-hand process is certainly independent of all other actions. The only successor with respect to that action is again state (s_0, t_0). A naive modification of the model checking algorithm of Fig. 5 would stop generating further states at that point, which is obviously inadequate.

Partial-order reduction algorithms [107], [54], [61], [42], [95] differ in how these problems are dealt with in order to arrive at a reasonably efficient algorithm that is adequate for the given task. The general idea is to approximate the semantic notion of commutativity of actions using syntactic criteria. For example, for a language based on shared variables, two actions of different processes are certainly independent if they do not update the same variable. For message passing communication, send and receive operations over the same channel are independent at those states where the channel is neither empty nor full. Second, the formula φ being analysed must be taken into account: call an action A *visible* for φ if A may change the value of a variable that occurs in φ. Holzmann and Peled [61] define an action to be *safe* if it is not visible and if it is provably independent (with the help of syntactic criteria) of all actions of different processes, even if these actions are currently disabled. The depth-first search algorithm shown in figure 5 can then be modified so that only successor states are considered for some process that can only perform safe actions at the current state. Consideration of the actions of

other processes is thus delayed. However, the delayed actions must be considered before a loop is completed. This rather simple heuristic can already lead to substantial savings and carries almost no overhead because the set of safe actions can be determined statically.

More elaborate reduction techniques are considered, for example, in [54], [94], [108]. There is always a tradeoff between the potential effectiveness of a reduction method and the overhead involved in computing a sufficient set of actions that must be explored at a given state. Moreover, the effectiveness of partial-order reductions in general depends on the structure of the system: while they are useless for tightly synchronized systems, they may dramatically reduce the numbers of states and transitions explored during model checking for loosely coupled, asynchronous systems.

Symmetry reductions are based on the observation that the behavior of systems, and the properties they satisfy, are often invariant under a permutation of certain parameters. For example, the precise data values transmitted in a protocol can be irrelevant if we are only interested in verifying if the message arrive in the original order. Similarly, process identities are irrelevant when verifying correctness properties of a resource allocator. In such situations, we may basically work with equivalence classes of states modulo a symmetry group rather than with full states, and in particular cut off the search (in the algorithm of Fig. 5) whenever we have already encountered an *equivalent*, rather than equal, pair in set `visited`. For more details, see [19], [64].

IV. Further topics

This survey has concentrated on the basic techniques for finite-state model checking. We conclude by mentioning some more recent developments and challenges for further research. Some of these issues are addressed in more detail in other contributions to this volume.

Abstraction: The infamous state explosion problem remains the most serious obstacle when applying model checking. It refers to the observation that the size of a system's state space is in general the product of the state spaces of its components, whereas the description only grows as the sum of the components' descriptions. We have already mentioned two ways to cope with this problem: *compression* techniques attempt to represent state spaces succinctly, as in symbolic model checking (BDDs) or by hashing state descriptions in explicit-state model checking. *Reduction* techniques try to counteract state space explosion by avoiding to explore the full product state space.

Still, the size of systems that can be analysed using model checking remains relatively limited: even seemingly large numbers such as 10^{100} states that have been handled successfully may be generated by systems with a few hundred bits of memory, which is a far cry from realistic hardware or software systems. Model checking must therefore be performed on rather abstract models. It is often advocated that model checking be applied to high-level designs during the early stages of system development because the payoff of finding bugs at that level is high whereas the costs are low, and methodological studies of integrating model checking in processes of system design can be helpful in this respect.

When the analysis of big models cannot be avoided, it is rarely necessary to consider them in full detail in order to verify or falsify some given property. This idea can be formalized as an abstraction function (or relation) that induces some abstract system model such that the property holds of the original, "concrete" model if it can be proven for the abstract model. (Dually, abstractions can be set up such that failure of the property in the abstract model implies failure in the concrete model.) In general, the appropriate abstraction relation depends on the application. Nevertheless, "push-button" techniques can be envisaged for specific classes of systems, say, hardware drivers of operating systems or process control software. Given a concrete model and an abstraction relation, one can either attempt to construct the abstract model using techniques of abstract interpretation [30] or verify the correctness of a proposed abstract model using theorem proving. Early literature on abstraction techniques include [22], [31], [79], [80].

A particularly attractive way of presenting abstractions is in the form of *predicate abstractions* where predicates of interest at the concrete level are mapped to Boolean variables at the abstract level. It is often possible to start with a very coarse abstraction of the system and apply the model checker to find an (abstract) counter-example to the property of interest. This counter-example must then be analyzed to determine whether it is feasible for the concrete system or not: the abstract model, by nature, contains transitions that have no counterpart in the original model. If the counter-example is feasible, it can be reported to the user and the process stops. Otherwise, the guard of the first transition that is not feasible provides an additional predicate that should be added to the set of predicates defining the abstraction relation, and the process starts again. This iterative approach is known as counter-example guided abstraction refinement (CEGAR) and has been successfully used for the verification of actual software. It is at the basis of tools such as BLAST [59] or SLAM [5], and currently constitutes one of the most active research topics in model checking.

Infinite-state systems: Throughout this exposition we have assumed the transition system to be finite-state. Of course, the explicit-state algorithms presented here can also be applied to infinite-state systems, but they may not terminate. There has been much research activity about model checking infinite-state (but finitely representable) systems, and we can do no more than to refer the interested reader to some survey papers on this very active field [18], [43], [44], [88]. In particular, pushdown systems have received much attention, as they can model executions of sequential recursive programs, and in particular make assertions about the contents of the recursion stack [11], [45].

Parameterized systems: One is often interested in the properties of a family of finite-state systems that differ in some parameter such as the number of processes. Although individual members of the family can be analyzed using standard model checking techniques, the verification of the entire family requires additional considerations. A natural idea

is to perform standard model checking for fixed parameter values and then establish correctness for arbitrary parameter values by induction. In some cases, even the induction step can be justified by model checking. For example, Browne et al. [12] suggest to model check a two-process system, and to establish a bisimulation relation between two-process and n-process systems, ensuring that formulas expressed in a suitable logic cannot distinguish between them. This approach has been extended in [75], [111] by using a finite-state process I that acts as an invariant in that the composition of I with another process is again bisimilar to I. Because both I and the individual processes are finite-state, this can be accomplished using (a variation of) standard model checking. Related techniques are described in [40], [51].

Compositional verification: The effects of state explosion can be mitigated when the overall verification effort can be reduced to considering the components of a complex system one at a time. As in the case of abstraction, compositional reasoning normally requires additional input from the user who must specify appropriate properties to be verified of the individual components. The main problem is that components cannot necessarily be expected to function correctly in arbitrary environments, because their design relies on properties of the system the components are expected to be part of. Thus, corresponding assumptions have to be introduced in the statement of the components' correctness properties. Early work on compositional verification [6], [96] required components to form a hierarchy with respect to their dependency. In general, however, every component is part of every other component's environment, and circular dependencies among components are to be expected. More recently, different formulations of assumption-commitment specifications have been studied [1], [28], [84] that can accomodate circular dependencies, based on a form of computational induction. A collection of papers on compositional methods for specification and verification is contained in [33]. Model checking algorithms for modular verification are described, among others, in [55], [66], [65].

Real-time systems: These constitute the main subject of this volume, and we only refer to the contributions in this volume on verification of real-time systems. Also, see Kwiatkowka's contribution for verification techniques concerning probabilistic systems.

REFERENCES

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[2] R. Alur and T. A. Henzinger. Logics and models of real time: a survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1992.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *38th IEEE Symposium on Foundations of Computer Science*, pages 100–109. IEEE Press, October 1997.

[4] A. Anuchitanukul. *Synthesis of Reactive Programs*. PhD thesis, Stanford University, 1995.

[5] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL 2002)*, pages 1–3, 2002.

[6] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *16th ACM Symp. on Theory of Computing*, pages 51–63. ACM Press, 1984.

[7] J. Bern, C. Meinel, and A. Slobodová. Global rebuilding of BDDs – avoiding the memory requirement maxima. In P. Wolper, editor, *7th Workshop on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 4–15. Springer-Verlag, 1995.

[8] A. Biere, A. Cimatti, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th ACM/IEEE Design Automation Conference (DAC'99)*, 1999.

[9] Armin Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen*. PhD thesis, Univ. Karlsruhe, Germany, 1997.

[10] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In R. Alur and T. Henzinger, editors, *8th Workshop on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1996.

[11] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper. An efficient automata approach to some problems on context-free grammars. *Information Processing Letters*, 74(5-6):221–227, 2000.

[12] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation*, 81:13–31, 1989.

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[14] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40(2):205–213, 1991.

[15] R. E. Bryant. Symbolic boolean manipulations with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–317, 1992.

[16] J. R. Büchi. On a decision method in restricted second-order arithmetics. In *International Congress on Logic, Method and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[18] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1997. http://www.elsevier.nl/locate/entcs/volume6.html.

[19] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1993.

[20] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10:47–71, 1997.

[21] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, N.Y., 1981. Springer-Verlag.

[22] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[23] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Mass., 1999.

[24] Edmund M. Clarke and Holger Schlingloff. Model checking. In A. Voronkov, editor, *Handbook of Automated Deduction*. Elsevier, 2000. To appear.

[25] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[26] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V.

[27] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[28] P. Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, 1994.

[29] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, 1:275–288, 1992.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[31] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods, and Calculi (PROCOMET '94)*, pages 561–581, Amsterdam, 1994. North Holland/Elsevier.

[32] M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260, Trento, Italy, 1999. Springer-Verlag.

[33] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[34] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE Intl. Conf. Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. http://sprout.stanford.edu/dill/murphi.html.

[35] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching time vs. linear time. *Journal of the ACM*, 33:151–178, 1986.

[36] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of μ-calculus. In C. Courcoubetis, editor, *5th Workshop on Computer-Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[37] E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *12th Symp. on Principles of Programming Languages (POPL'85)*, New Orleans, 1985. ACM Press.

[38] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *1st Symp. on Logic in Computer Science*, Boston, Mass., 1986. IEEE Press.

[39] E. Allen Emerson. *Handbook of theoretical computer science*, chapter Temporal and modal logic, pages 997–1071. Elsevier Science Publishers B.V., 1990.

[40] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV'96)*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[41] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking. *Distributed Computing*, 6:155–164, 1993.

[42] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[43] J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[44] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *14th IEEE Symposium on Logic in Computer Science*, pages 352–359, Trento, Italy, 1999. IEEE Press.

[45] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[46] E. Felt, G. York, R. Brayton, and A. S. Vincentelli. Dynamic variable reordering for BDD minimization. In *European Design Automation Conference*, pages 130–135, 1993.

[47] H. Fuji, G. Oomoto, and C. Hori. Interleaving based variable ordering methods for binary decision diagrams. In *Intl. Conf. on Computer Aided Design (ICCAD'93)*. IEEE Press, 1993.

[48] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Clarendon Press, Oxford, UK, 1994.

[49] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *13th Intl. Conf. Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65, Paris, France, 2001. Springer-Verlag.

[50] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes LTL verification more efficient. In K. Jensen and A. Podelski, editors, *10th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219, Barcelona, Spain, 2004. Springer-Verlag.

[51] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

[52] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[53] P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. In *11th Ann. IEEE Symp. on Logic in Computer Science (LICS'96)*, New Brunswick, NJ, 1996. IEEE Press.

[54] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[55] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[56] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs and Lenore Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 191–205, Edinburgh, Scotland, April 2005. Springer-Verlag.

[57] Sergiu Hart and Micha Sharir. Probabilistic temporal logics for finite and bounded models. In *ACM Symp. Theory of Computing (STOC'84)*, pages 1–13, New York, NY, USA, 1984. ACM Press.

[58] Thomas A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43:135–141, September 1992.

[59] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth McMillan. Abstractions from proofs. In *31st Annual Symp. Princ. of Prog. Lang. (POPL 2004)*, pages 232–244. ACM Press, 2004.

[60] Gerard Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, November 1998.

[61] Gerard Holzmann and Doron Peled. An improvement in formal verification. In *IFIP WG 6.1 Conference on Formal Description Techniques*, pages 197–214, Bern, Switzerland, 1994. Chapman & Hall.

[62] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Reading, Mass., 1979.

[63] Michael Huth and Mark D. Ryan. *Logic in Computer Science*. Cambridge University Press, Cambridge, U.K., 2000.

[64] C. N. Ip and D. L. Dill. State reduction using reversible rules. In *33rd Conf. Design Automation*, pages 564–567, Las Vegas, NV, 1996. ACM Press.

[65] Bernhard Josko. Verifying the correctness of AADL modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlag, Berlin, 1989.

[66] Bernhard Josko. *Modular Specification and Verification of Reactive Systems*. PhD thesis, Univ. Oldenburg, Fachbereich Informatik, April 1993.

[67] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Univ. of California at Los Angeles, 1968.

[68] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[69] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[70] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.

[71] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *6th Intl. Conf. on Computer-Aided Verification (CAV'94)*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Full version (1999) available at http://www.cs.rice.edu/~vardi/papers/.

[72] O. Kupferman and M. Y. Vardi. Verification of fair transition systems. In R. Alur and T. Henzinger, editors, *8th Workshop on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 372–382. Springer-Verlag, 1996.

[73] O. Kupferman and M.Y. Vardi. Complementation constructions for nondeterministic automata on infinite words. In Nicolas Halbwachs and Lenore Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 206–221, Edinburgh, Scotland, 2005. Springer-Verlag.

[74] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not so weak. In *5th Israeli Symposium on Theory of Computing and Systems*, pages 147–158. IEEE Press, 1997.

[75] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *8th Ann. ACM Symp. on Principles of Distributed Computing*. ACM Press, 1989.

[76] Leslie Lamport. 'sometime' is sometimes 'not never'. In *Proc. 7th Ann. Symp. on Princ. of Prog. Lang. (POPL'80)*, pages 174–185. ACM SIGACT-SIGPLAN, January 1980.

[77] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392, Copenhague, Denmark, 2002. IEEE Computer Society Press.

[78] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, Berlin, June 1985. Springer-Verlag.

[79] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[80] D. E. Long. *Model checking, Abstraction and Compositional Verification*. PhD thesis, CMU School of Computer Science, 1993. CMU-CS-93-178.

[81] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *9th. ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM, 1990.

[82] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Specification*. Springer-Verlag, New York, 1992.

[83] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Safety properties*. Springer-Verlag, New York, 1995.

[84] Kenneth L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35, Haifa, Israel, 1997. Springer-Verlag.

[85] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[86] R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, Cambridge, Mass., 1971.

[87] Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[88] Faron Moller. Infinite results. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216, Pisa, Italy, 1996. Springer-Verlag.

[89] C. Morgan and A. K. McIver. An expectation-transformer model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7(6):779–804, 1999.

[90] D. E. Muller. Infinite sequences and finite machines. In *Switching Circuit Theory and Logical Design: Fourth Annual Symposium*, pages 3–16, New York, 1963. IEEE Press.

[91] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *13th ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 275–283. Springer-Verlag, 1986.

[92] D.E. Muller, A. Saoudi, and P.E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *3rd IEEE Symposium on Logic in Computer Science*, pages 422–427. IEEE Press, 1988.

[93] D. M. Park. Finiteness is mu-ineffable. Theory of Computation Report 3, University of Warwick, 1974.

[94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[95] W. Penczek, R. Gerth, and R. Kuiper. Partial order reductions preserving simulations. Submitted for publication, 1999.

[96] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F 13 of *ASI*, pages 123–144. Springer-Verlag, Berlin, 1985.

[97] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[98] Shmuel Safra. On the complexity of ω-automata. In *29th IEEE Symposium on Foundations of Computer Science*, pages 319–327. IEEE Press, 1988.

[99] Klaus Schneider. Yet another look at LTL model checking. In *IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, Lecture Notes in Computer Science, Bad Herrenalb, Germany, 1999.

[100] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190, Edinburgh, UK, 2005. Springer-Verlag.

[101] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.

[102] Colin Stirling. *Handbook of Logic in Computer Science*, volume 2, chapter Modal and temporal logics, pages 477–563. Oxford Science Publications, Clarendon Press, Oxford, 1992.

[103] Colin Stirling. Bisimulation, model checking, and other games. Mathfit instructional meeting on games and computation, 1997.

[104] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[105] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*, pages 133–194. Elsevier, Amsterdam, 1990.

[106] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1997.

[107] A. Valmari. A stubborn attack on state explosion. In *2nd International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer-Verlag.

[108] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[109] Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1995.

[110] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[111] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Intl. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[112] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–93, 1983.

Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems *

Vlad Rusu Hervé Marchand Thierry Jéron

Irisa/Inria Rennes, Campus de Beaulieu, 35042 Rennes Cedex
First.Last@irisa.fr

Abstract

This paper presents a combination of verification and conformance testing techniques for the formal validation of reactive systems. A formal specification of a system, which may be infinite-state, and a set of safety properties are assumed. Each property is verified on the specification using automatic techniques based on abstract interpretation, which are sound, but, as a price to pay for automation, are not necessarily complete. Next, for each property, a test case is automatically generated from the specification and the property, and is executed on a black-box implementation of the system to detect violations of the property by the implementation and non-conformances between implementation and specification. If the verification step did not conclude, the test execution may also detect violations of the property by the specification.

Keywords: verification, conformance testing, test generation

1. Introduction

Formal verification and conformance testing are two well-established approaches for validating reactive systems. Both approaches consist in checking the consistency between two representations of a system:

- formal verification typically compares a formal *specification* of the system with respect to some higher-level required *properties*;
- conformance testing [1, 5] compares the observable behaviour of a black-box *implementation* of the system with that described by the specification.

A formal validation chain for reactive systems, combining verification and conformance testing, may naturally consists of the following steps:

1. the properties are automatically verified on the specification;
2. test cases are automatically derived from the specification and the properties;
3. the test cases are executed on the black-box implementation of the system, to check the satisfaction of the properties by the implementation and the conformance between implementation and specification.

In this paper we formally define and study such a validation chain. We consider a general class of specifications which may be infinite-state (automata extended with variables, which communicate with the environment by means of inputs and outputs carrying parameters). In this setting, the verification step (in particular, for safety properties) is undecidable. In order to keep it automatic and ensure that it always terminates, we adopt approximate, conservative verification techniques based on abstract interpretation [7], which may either prove the property, or terminate with a "don't know" answer.

The main contribution of the paper lies in the second step of the proposed validation chain. It is a test generation algorithm that takes into account the infinite-state nature of the specifications and the incompleteness of the verification step. The algorithm takes as inputs a specification and a safety property, and produces a test case for checking the conformance between a given implementation and the specification, and the satisfaction of the safety property by the implementation. To deal with infinite-state specifications and properties, the algorithm is *symbolic*: it does not attempt to enumerate the (potentially infinite) domain of the specification's variables, but deals with the variables by means of symbolic computations. As a consequence of the incompleteness of the verification step, the test cases generated by our algorithm may also detect violations of the property by the *specification* when executed on the *implementation*. Hence, test execution may detect one or several of the following inconsistencies:

- violation of the property by the specification,
- violation of the property by the implementation,
- violation of conformance between implementation and specification.

These results are returned to the user in the form of test verdicts, and may be employed to fix errors in the implementation, specification, or the properties.

*This paper also appeared in *Formal Methods'05*, LNCS 3582.

The rest of the paper is organised as follows. Section 2 presents the model of Input-Output Symbolic Transition Systems (IOSTS) and, in Section 3 we set the framework for verification and testing using IOSTS as the underlying model.

Section 4 defines our symbolic test generation algorithm. The algorithm is proved correct, in the sense that the verdicts returned by test execution correctly characterise the relations between implementation, specification, and property. Moreover, the (infinite) set of all test cases generated in this manner may, in principle, discover all implementations that do not conform to a given specification according to the standard **ioco** relation [19].

As a by-product of the correctness proofs, we show that **ioco**-conformance with respect to a given specification is a safety property. We also provide a symbolic construction of the canonical tester [4] for **ioco**-conformance with respect to a given specification.

Section 5 outlines a technique for optimising test cases towards detecting the violation of the property. We show that this optimisation preserves the correctness of the test verdicts. The overall approach is illustrated on a simple example. The full version of this paper [17] contains a larger example (the Bounded Retransmission Protocol [11]) and provides proofs of the results.

2. The IOSTS Model

The model of Input-Output Symbolic Transition Systems (IOSTS) is inspired from I/O automata [15]. Unlike I/O automata, IOSTS do not require *input-completeness* (all input actions do not need to be enabled all the time).

Definition 1 (IOSTS) *An IOSTS is a tuple* $\langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ *where*

- D *is a finite set of typed* Data*, partitioned into a set* V *of* variables *and a set* P *of* parameters*. For* $d \in D$*,* $type(d)$ *denotes the type of* d*.*
- Θ *is the* initial condition*, a Boolean expression on* V*,*
- Q *is a nonempty, finite set of* locations *and* $q^0 \in Q$ *is the* initial location*.*
- Σ *is a nonempty, finite* alphabet*, which is the disjoint union of a set* $\Sigma^?$ *of* input actions *and a set* $\Sigma^!$ *of* output actions[1]*. For each action* $a \in \Sigma$*, its* signature $sig(a) = \langle p_1, \ldots, p_k \rangle \in P^k$ $(k \in \mathbb{N})$ *is a tuple of parameters.*
- $\mathcal{T}$ *is a set of* transitions*. Each transition is a tuple* $\langle q, a, G, A, q' \rangle$ *made of:*
  - *a location* $q \in Q$*, called the* origin *of the transition.*
  - *an action* $a \in \Sigma$ *called the* action *of the transition.*
  - *a Boolean expression* G *on* $V \cup sig(a)$*, called the* guard*.*
  - *an* assignment A*, which is a set of expressions of the form* $(x := A^x)_{x \in V}$ *such that, for each* $x \in V$*, the right-hand side* A^x *of the assignment* $x := A^x$ *is an expression on* $V \cup sig(a)$*.*
  - *a location* $q' \in Q$ *called the* destination *of the transition.*

A simple example of IOSTS is depicted in Figure 1. This system expects a $START$ input carrying an integer parameter p, and saves the value of p into the variable x. Then, as long as x is strictly positive, its value is emitted to the environment via the output MSG carrying the parameter m. The variable x is decreased by 1, and when it reaches 0, the $STOP$ output is emitted.

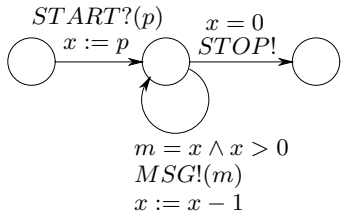


Figure 1. sample IOSTS $\mathcal{S}$

Semantics.

The semantics of IOSTS is described in terms of input-output labelled transitions systems (IOLTS).

Definition 2 *An IOLTS is a tuple* $\langle S, S^0, \Lambda, \rightarrow \rangle$ *where* S *is a set of* states*, which may be infinite,* $S^0 \subseteq S$ *is the set of* initial states*,* $\Lambda = \Lambda^? \cup \Lambda^!$ *is a set of (input or output)* actions*, and* $\rightarrow \subseteq S \times \Lambda \times S$ *is the* transition relation*.*

Intuitively, the IOLTS semantics of an IOSTS $\langle D = V \cup P, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ enumerates of the possible tuples of values (hereafter called *valuations*) of parameters P and variables V. Let $\mathcal{V}$ denote the set of valuations of the variables V, and Π denote the set of valuations of the parameters P. Then, for an expression E involving (a subset of) $V \cup P$, and for $\nu \in \mathcal{V}$, $\pi \in \Pi$, we denote by $E(\nu, \pi)$ the value obtained by substituting in E each variable by its value according to ν, and each parameter by its value according to π. For $P' \subseteq P$, we denote by $\Pi_{P'}$ the restriction of the set Π of valuations to the set P' of parameters.

Definition 3 *The semantics of an IOSTS* $\mathcal{S} = \langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ *is an IOLTS* $[\![\mathcal{S}]\!] = \langle S, S^0, \Lambda, \rightarrow \rangle$*, defined as follows:*

- *the set of states is* $S = Q \times \mathcal{V}$*,*
- *the set of initial states is* $S^0 = Q \times \mathcal{V}^0$*,*

[1] For simplicity, only input and output actions are considered here. A more detailed model, which also contains internal actions, is defined in the full paper [17].

- *the set of actions* $\Lambda = \{\langle a, \pi\rangle | a \in \Sigma, \pi \in \Pi_{sig(a)}\}$, *also called the set of* valued actions, *is partitioned into the sets* $\Lambda^?$ *of* valued inputs *and* $\Lambda^!$ *of* valued outputs, *such that for* $\# \in \{?, !\}$, $\Lambda^\# = \{\langle a, \pi\rangle | a \in \Sigma^\#, \pi \in \Pi_{sig(a)}\}$.
- $\rightarrow$ *is the smallest relation in* $S \times \Lambda \times S$ *defined by the following rule:*

$$\frac{\langle q,\nu\rangle, \langle q',\nu'\rangle \in S,\ \langle a,\pi\rangle \in \Lambda,\ \langle q,a,G,A,q'\rangle \in \mathcal{T},, G(\nu,\pi) = true,\ \nu' = A(\nu,\pi)}{\langle q,\nu\rangle \overset{\langle a,\pi\rangle}{\rightarrow} \langle q',\nu'\rangle}$$

The rule says that the valued action $\langle a, \pi\rangle$ takes the system from a state $\langle q, \nu\rangle$ to a state $\langle q', \nu'\rangle$ if there exists a transition $t = \langle q, a, G, A, q'\rangle$ whose guard G evaluates to $true$ when the variables evaluate according to ν and the parameters carried by the action a evaluate according to π. Then, the assignment A of the transition maps the pair (ν, π) to ν'.

Definition 4 (run) *A run fragment is a sequence of alternating states and valued actions* $\beta : \quad s_1 \overset{\alpha_1}{\rightarrow} s_2 \overset{\alpha_2}{\rightarrow} \cdots s_{n-1} \overset{\alpha_{n-1}}{\rightarrow} s_n$. *A* run *is a run fragment that starts in an initial state.*

A state is *reachable* if it is the last state of a run. For a sequence $\sigma = \alpha_1\alpha_2 \cdots \alpha_n$ of valued actions, we sometimes write $s \overset{\sigma}{\rightarrow} s'$ for $\exists s_1, \ldots s_{n+1} \in S.\ s = s_1 \overset{\alpha_1}{\rightarrow} s_2 \overset{\alpha_2}{\rightarrow} \cdots s_n \overset{\alpha_n}{\rightarrow} s_{n+1} = s'$. For a set of states $S' \subseteq S$ of the IOSTS we write $s \overset{\sigma}{\rightarrow} S'$ if there exists a state $s' \in S'$ such that $s \overset{\sigma}{\rightarrow} s'$.

Definition 5 (trace) *The* trace *of a run* ρ *is the projection of* ρ *on* $\Lambda^! \cup \Lambda^?$. *The set of traces of an IOSTS* $\mathcal{S}$ *is denoted by* $\mathit{Traces}(\mathcal{S})$.

Let $F \subseteq Q$ be a set of locations of an IOSTS $\mathcal{S}$. A run ρ is *recognised by* F if it ends in a state in $F \times \mathcal{V}$. A trace is *recognised by* F if it is the projection on $\Lambda^! \cup \Lambda^?$ of a recognised run. The set of recognised traces is denoted by $\mathit{RTraces}(\mathcal{S}, F)$.

An IOSTS is *deterministic* if in each location, the guards of the transitions labelled by the same action are mutually exclusive. All the IOSTS considered in this paper are deterministic. In the full version [17], more general IOSTS are also considered (nondeterministic IOSTS with internal actions). A symbolic *determinisation* operation, which consists in transforming a nondeterministic IOSTS into a deterministic one having the same set of traces, is also presented. The operation is proved correct and terminates for a subclass of IOSTS [20].

3. Verification and conformance testing with IOSTS

This section sets the framework for verification and conformance testing with IOSTS. First, we present a few operations on IOSTS, and then the satisfaction relation and the conformance relation between IOSTS are formally defined.

3.1. Parallel Product

The *parallel product* of two IOSTS is an IOSTS whose set of traces traces (resp. recognised traces) are the intersection of the set of traces (resp. recognised traces) of the operands. This operation imposes that the IOSTS have no shared variables, but are defined on the same alphabets of actions and same parameters.

Definition 6 (Compatible IOSTS) *For* $j = 1, 2$, *the two IOSTS* $\mathcal{S}_j = \langle D_j = V_j \cup P_j, \Theta_j, Q_j, q_j^0, \Sigma_j, \mathcal{T}_j\rangle$ *with data* D_j *and alphabet* $\Sigma_j = \Sigma_j^? \cup \Sigma_j^!$ *are* compatible *if* $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma_1^! = \Sigma_2^!$, $\Sigma_1^? = \Sigma_2^?$.

Definition 7 (Parallel Product) *The parallel product* $\mathcal{S} = \mathcal{S}_1||\mathcal{S}_2$ *of two compatible IOSTS* $\mathcal{S}_1, \mathcal{S}_2$ *is the IOSTS* $\langle D, P, \Theta, Q, q^0, \Sigma, \mathcal{T}\rangle$ *that consists of the following elements:* $V = V_1 \cup V_2$, $P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $Q = Q_1 \times Q_2$, $q^0 = \langle q_1^0, q_2^0\rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$ *The set* $\mathcal{T}$ *of transitions of the composed system is the smallest set defined by the rule:*

$$\frac{\langle q_1, a, G_1, A_1, q_1'\rangle \in \mathcal{T}_1 \quad \langle q_2, a, G_2, A_2, q_2'\rangle \in \mathcal{T}_2}{\langle\langle q_1, q_2\rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle q_1', q_2'\rangle\rangle \in \mathcal{T}}$$

Lemma 1 (traces of the parallel product) $\mathit{Traces}(\mathcal{S}_1||\mathcal{S}_2) = \mathit{Traces}(\mathcal{S}_1) \cap \mathit{Traces}(\mathcal{S}_2)$; $\mathit{RTraces}(\mathcal{S}_1||\mathcal{S}_2, F_1 \times F_2) = \mathit{RTraces}(\mathcal{S}_1, F_1) \cap \mathit{RTraces}(\mathcal{S}_2, F2)$.

3.2. Quiescence, and suspension IOSTS

In conformance testing it is assumed that the environment may observe not only outputs, but also *absence of outputs* (i.e., in a given state, the system does not emit any output for the environment to observe). This is called *quiescence* in conformance testing [19]. On a black-box implementation, quiescence is observed using timers: a timer is reset whenever the environment sends a stimulus to the implementation; when the timer expires, the environment observes quiescence.

In order to distinguish a quiescence that is also present in a specification from one that is not, quiescence can be made explicit on a specification by a symbolic operation called *suspension*. This operation transforms an IOSTS $\mathcal{S}$ into an IOSTS $\mathcal{S}^\delta$, also called the *suspension IOSTS* of $\mathcal{S}$. Each location q of $\mathcal{S}^\delta$ contains a new self-looping transition, labelled with a new output action δ, which may be fired if and only if no other output action may be fired in q. Formally,

Definition 8 (Suspension) *Given* $\mathcal{S} = \langle D = V \cup P, \Theta, Q, q^0, \Sigma = \Sigma^! \cup \Sigma^?, \mathcal{T}\rangle$ *an IOSTS, the* suspension *IOSTS* $\mathcal{S}^\delta$ *is the tuple* $\langle D = V \cup P, \Theta, Q, q^0, (\Sigma^! \cup \{\delta\}) \cup \Sigma^?, \mathcal{T} \cup \bigcup_{q \in Q} \langle q, \delta, G_{\delta,q}, (v := v)_{v \in V}, q\rangle\rangle$ *where*

$$G_{\delta,q}: \quad \bigwedge_{a\in\Sigma^!} \neg G_{a,q}$$

and

$$G_{a,q}: \quad \bigvee_{t=\langle q,a,G,A,q'\rangle\in\mathcal{T}} \exists sig(a).G.$$

For the IOSTS $\mathcal{S}$ depicted in Figure 1, the IOSTS $\mathcal{S}^\delta$ is depicted in Figure 2. The guard $x < 0$ of the transition labeled δ is obtained by simplifying the expression $\neg(x = 0 \vee \exists m, m = x \wedge x > 0)$, which corresponds to Formula (1) above.

In this system, a *START* input with a negative parameter ($p < 0$) does not allow for *MSG* or *STOP* outputs, i.e., the system is quiescent after *START*. This is made explicit by the special output $\delta!$ after *START*.

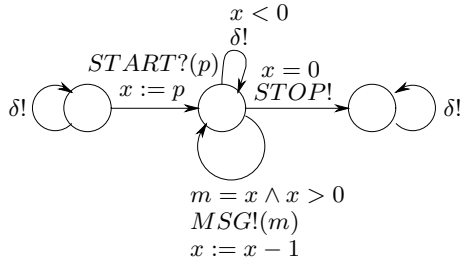


Figure 2. Suspension IOSTS $\mathcal{S}^\delta$

3.3. Verification of Safety Properties

The problem considered here is: given a reactive system modelled by an IOSTS $\mathcal{S}$, and a safety property ψ defined on its traces, does $\mathcal{S}$ satisfy ψ? We model safety properties using *observers*, which are deterministic IOSTS equipped with a set of "bad" locations; the property is violated when a "bad" location is reached.

Definition 9 (Observer) *An observer is a deterministic IOSTS ω together with a set of dedicated locations Violate$_\omega \subseteq Q_\omega$, which are deadloocks (no outgoing transitions). An observer $(\omega, \textit{Violate}_\omega)$ is compatible with an IOSTS M if ω is compatible with M. The set of observers compatible with M is denoted $\Omega(M)$.*

An observer $\omega \in \Omega(M)$ defines a safety property on $(\Lambda^!_M \cup \Lambda^?_M)^*$, namely, the property that is satisfied by all sequences in $(\Lambda^!_M \cup \Lambda^?_M)^* \setminus \textit{RTraces}(\omega, Violate_\omega)$ (and those sequences only). In particular, if M is the suspension IOSTS $\mathcal{S}^\delta$ of a given IOSTS $\mathcal{S}$, then the property is satisfied by a subset of $(\Lambda^!_s \cup \{\delta\} \cup \Lambda^?_s)^*$.

For example the observer ω_1 depicted in Figure 3 describes the safety property which says that between *START* input carrying a parameter $p > 0$, and a *STOP* output, the system must exhibit at least one *MSG* output. The set of "bad" locations is $\{\textit{Violate}\}$. The self-loops "*" denote all actions (including the quiescence δ) that do not label other outgoing transitions. The observer ω_2 also

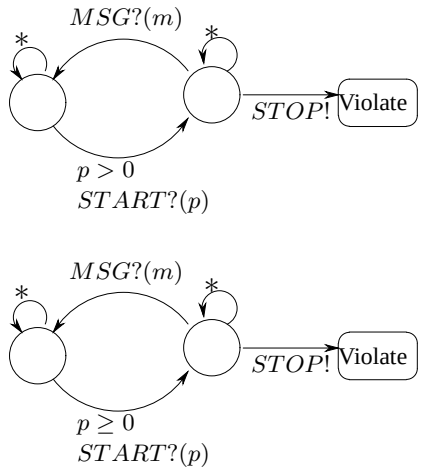


Figure 3. Observers : ω_1 (top), ω_2 (bottom).

depicted in Figure 3 describes almost the same property except for the fact that *START* carries a parameter $p \geq 0$.

An IOSTS astisfies an observer if no trace of the IOSTS is recognised by the observer:

Definition 10 (IOSTS Satisfies Observer) *For an IOSTS $\mathcal{S}$ and an observer $(\omega, \textit{Violate}_\omega) \in \Omega(\mathcal{S})$, we say that $\mathcal{S}$ satisfies $(\omega, \textit{Violate}_\omega)$, denoted by $\mathcal{S} \models (\omega, \textit{Violate}_\omega)$, if $\textit{Traces}(\mathcal{S}) \cap \textit{RTraces}(\omega, \textit{Violate}_\omega) = \emptyset$.*

Let Q denote the set of locations of $\mathcal{S}$. Then, $\textit{Traces}(\mathcal{S}) = \textit{RTraces}(\mathcal{S}, Q)$ and $\textit{RTraces}(\mathcal{S}\|\omega, Q \times \textit{Violate}_\omega) = \textit{RTraces}(\mathcal{S}, Q) \cap \textit{RTraces}(\omega, \textit{Violate}_\omega)$ (cf. Lemma 1). Hence, checking $\mathcal{S} \models (\omega, \textit{Violate}_\omega)$ amounts to checking the emptyness of the set $\textit{RTraces}(\mathcal{S}\|\omega, Q \times \textit{Violate}_\omega)$. This can be done checking that the intersection between the set of *reachable states* of $\mathcal{S}\|\omega$, and the set of states whose locations lie in $Q \times \textit{Violate}_\omega$, is empty. Alternatively, the intersection between the set of states from which $Q \times \textit{Violate}_\omega$ is reachable (also called the *coreachable set* of $Q \times \textit{Violate}_\omega$), and the set of initial states, can be checked for emptyness.

However, reachable and coreachable sets are not computable in general because of undecidability problems. Approximate analysis techniques such as abstract interpretation [7], can be used to compute over-approximations of them.

Our tool STG (Symbolic Test Generation) [6] is interfaced with a tool called NBac [13] for this purpose. First, STG automatically computes the product $\omega||\mathcal{S}$, and then, NBac automatically performs an approximate reachability analysis (from the initial states) and approximate coreachability analysis (to the violating locations) of the product. These tools can be employed to prove, e.g., that the IOSTS $\mathcal{S}^\delta$ depicted in Figure 1 does satisfy the observer ω_1 depicted in Figure 3. (The violating locations are found unreachable, hence, the property holds).

On the other hand, it is impossible *in general* to prove automatically that an IOSTS does *not* satisfy an observer. Such a situation occurs with the IOSTS $\mathcal{S}^\delta$ in Figure 1 and the observer ω_2 depicted in the right-hand side of Figure 3:

$\mathcal{S}^\delta$ does not satisfy ω_2, because a *START* input carrying the parameter $p = 0$ allows for a *STOP* output to be emitted (without any MSG inputs in between), which violates the property of interest (the *Violate* location is reached).

Combining observers. The parallel product of two observers $(\omega, \mathit{Violate}_\omega)$ and $(\varphi, \mathit{Violate}_\varphi)$ can be also interpreted in terms of safety properties. We use these properties in Section 4. A natural choice is to equip the product $\omega||\varphi$ with the set of locations $\mathit{Violate}_\omega \times \mathit{Violate}_\varphi$; by Lemma 1, $\mathit{RTraces}(\omega||\varphi, \mathit{Violate}_\omega \times \mathit{Violate}_\varphi) = \mathit{RTraces}(\omega, \mathit{Violate}_\omega) \cap \mathit{RTraces}(\varphi, \mathit{Violate}_\varphi)$; hence, we obtain a safety property which is violated whenever both safety properties described by $(\omega, \mathit{Violate}_\omega)$ and $(\varphi, \mathit{Violate}_\varphi)$ are violated. Alternative choices for the violating locations are, e.g., $\mathit{Violate}_\omega \times (Q_\varphi \setminus \mathit{Violate}_\varphi)$, which indicates the violation of the former property, but not that of the latter; and, $(Q_\omega \setminus \mathit{Violate}_\omega) \times \mathit{Violate}_\varphi$, which indicates the violation of the latter property, but not of the former.

3.4. Conformance Testing

A *conformance relation* formalises the set of implementations that behave consistently with a specification. An implementation $\mathcal{I}$ is not a formal object (it is a physical system) but, in order to reason about conformance, it is necessary to assume that the semantics of $\mathcal{I}$ can be modelled by a formal object. We assume here that it is modelled by an IOLTS (cf. Definition 2). The notions of trace and quiescence are defined for IOLTS just as for IOSTS. The implementation is also assumed to be *input-complete*, i.e., all its inputs are enabled in all states.

These assumptions are called *test hypothesis* in conformance testing. The central notion in conformance testing is that of *conformance relation*; the standard **ioco** relation defined by Tretmans [19] can be rephrased as

Definition 11 (ioco conformance) *An implementation $\mathcal{I}$* **ioco***-conforms to a specification $\mathcal{S}$, denoted by $\mathcal{I}$* **ioco** *$\mathcal{S}$, if $\mathit{Traces}(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \cap Traces(\mathcal{I}^\delta) \subseteq \mathit{Traces}(\mathcal{S}^\delta)$.*

Intuitively, an implementation $\mathcal{I}$ **ioco**-conforms to its specification $\mathcal{S}$, if, after each trace of the suspension IOSTS $\mathcal{S}^\delta$, the implementation only exhibits outputs and quiescences allowed by $\mathcal{S}^\delta$. Hence, in this framework, the specification is *partial* with respect to inputs, i.e., after an input that is not described by the specification, the implementation may have any behaviour, without violating conformance to the specification. This corresponds to the intuition that a specification models a given set of services that must be provided by a system; a particular implementation of the system may implement more services than specified, but these additional features should not influence its conformance.

Example. An implementation that exhibits the trace $START?(1) \cdot STOP!$ does not conform to the specification $\mathcal{S}$ depicted in Figure 1 - this trace is not present in the IOSTS $\mathcal{S}^\delta$ (Figure 2). For the same reason, the trace $START?(1) \cdot \delta!$ reveals a non-conformance to $\mathcal{S}$. On the other hand, a trace such as $START?(1) \cdot START?(1) \cdot STOP!$ does not pose problems for conformance, as $\mathcal{S}^\delta$ does not constrain the traces of the system after the second *START?* in any way.

4. Testing for Safety and Conformance

This section shows how to generate a test case from a specification using a safety property as a guide. The test case attempts to detect violations of the property by an implementation of the system and violations of the conformance between the implementation and the specification. Moreover, if the verification step (Section 3.3) could not establish the fact that the specification satisfies the property, the generated test cases may also detect violations of the property by the specification when executed on the implementation.

We show that the test cases generated by our method always return correct verdicts. In this sense, the test generation method itself is correct.

Outline. We first define the *output-completion* $\Sigma^!(M)$ of a deterministic IOSTS M. We then show that the output-completion of the IOSTS of $\mathcal{S}^\delta$ is a *canonical tester* [4] for $\mathcal{S}$ and the **ioco** relation defined in Section 3.4 (a canonical tester for a specification with respect to a given relation allows, in principle, to detect every implementation that disagrees with the specification according to the relation). This derives from the fact, stated in Lemma 2 below, that **ioco**-conformance to a specification $\mathcal{S}$ is equivalent to satisfying (a safety property described by) an observer obtained from $\Sigma^!(\mathcal{S}^\delta)$.

Finally, by composing this observer with another observer $(\omega, \mathit{Violate}_\omega)$ we obtain test cases for checking the conformance to $\mathcal{S}$ and the satisfaction of $(\omega, \mathit{Violate}_\omega)$.

Definition 12 (Output-completion) *Given a deterministic IOSTS $M = \langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$, the* output completion *of M is the IOSTS*

$$\Sigma^!(M) = \langle D, \Theta, Q \cup \{\mathit{Fail}\}, q^0, \Sigma, \mathcal{T} \cup \bigcup_{q \in Q, a \in \Sigma^!} \langle q, a, \bigwedge_{\langle q,a,G_t,A_t,q'_t \rangle \in \mathcal{T}} \neg G_t, Id_V \mathit{Fail} \rangle \rangle$$

where Id_V is the identity assignment $(x := x)_{x \in V}$.

Interpretation: $\Sigma^!(M)$ is obtained from M by adding a new location $\mathit{Fail}_M \notin Q$, and for each $q \in Q$ and $a \in \Sigma^!$, a transition with origin q, destination Fail_M, action a, identity assignments and guard $\bigwedge_{t=\langle q,a,G_t,A_t,q'_t \rangle \in \mathcal{T}} \neg G_t$.

Hence, any output not fireable in M becomes fireable in $\Sigma^!(M)$ and leads to the new (deadlock) location Fail_M.

The output-completion of an IOSTS M can be seen as an observer, by choosing $\{\mathit{Fail}_M\}$ as the set of violating locations. The following lemma says that conformance to a specification $\mathcal{S}$ is a safety property, namely, the property whose negation is represented by the observer $(\Sigma^!(\mathcal{S}^\delta), \{Fail_{\mathcal{S}^\delta}\})$.

Lemma 2 $\mathcal{I}$ **ioco** $\mathcal{S}$ *iff* $\mathcal{I}^\delta \models (\Sigma^!(\mathcal{S}^\delta), \{Fail_{\mathcal{S}^\delta}\}))$.

The lemma also says that the IOSTS $\Sigma^!(\mathcal{S}^\delta)$ is a canonical tester for **ioco**-conformance to $\mathcal{S}$. Indeed, $\mathcal{I}^\delta \models (\Sigma^!(\mathcal{S}^\delta), \{Fail_{\mathcal{S}^\delta)}\}$ can be interpreted as the fact that execution of $\Sigma^!(\mathcal{S}^\delta)$ on the implementation $\mathcal{I}$ never leads to a "Fail" verdict; the fact that this is equivalent to $\mathcal{I}$ **ioco** $\mathcal{S}$ (as stated by Lemma 2) amounts to having a canonical tester [4].

A canonical tester is, in principle, enough for detecting all implementations that do not conform to a given specification. However, our goal in this paper is to detect, in addition to such non-conformances, other potential violations of other (additional) safety properties coming from, e.g., the system's requirements.

The observers (cf. Definition 9) employed for expressing such properties also serve as a test selection mechanism; by Lemma 1, the product between an observer and the canonical tester can be used to define a subset of traces of interest among the many possible traces of the canonical tester.

We first note that for an IOSTS M and an observer $(\omega, \mathit{Violate}_\omega) \in \Omega(M)$, the IOSTS $\omega||\Sigma^!(M)$ can be interpreted as an observer of M by choosing its set of violating locations. Let for now this set be $\{\mathit{Fail}_M\} \times \mathit{Violate}_\omega$, denoted by $\mathit{ViolateFail}_{\omega||\Sigma^!(M)}$.The subscript is omitted whenever it is clear from the context.

Definition 13 *For* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$, $\mathit{test}(\mathcal{S}, \omega) \triangleq \omega||\Sigma^!(\mathcal{S}^\delta)$.

In the rest of the section we show that every such $\mathit{test}(\mathcal{S}, \omega)$ can be seen as a test case that refines the canonical tester, as violations of $(\omega, \mathit{Violate}_\omega)$ are also checked.

Proposition 1 $\mathcal{I}$ **ioco** $\mathcal{S}$ *iff* $\forall(\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta)$. $\mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)})$.

Interpretation. The IOSTS $\mathit{test}(\mathcal{S}, \omega)$ can be seen as a test case to be executed in parallel with an implementation $\mathcal{I}$. Proposition 1 says that if this execution enters a location in $\mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)}$ (= $\mathit{Violate}_\omega \times \{\mathit{Fail}_{\mathcal{S}^\delta}\}$), then the implementation violates both the property defined by $(\omega, \mathit{Violate}_\omega)$ and the conformance to specification $\mathcal{S}$. In this situation, the ***ViolateFail*** verdict is given:

ViolateFail: the implementation violates both the property *and* the conformance.

The proposition also says that the (infnite) set $\{\mathit{test}(\mathcal{S}, \omega) | (\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta)\}$ of test cases is "exhaustive" for checking **ioco**-conformance to a given specification $\mathcal{S}$, meaning that all non-conformances may, in principle, be detected.

We now consider another interpretation of the IOSTS $\omega||\Sigma^!(M)$, which leads to another test verdict. Choosing the violating locations to be $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_M\}$ results in a different observer. We denote by $\mathit{Fail}_{\omega||\Sigma^!(M)}$ the set $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_M\}$. The subscript is omitted whenever the context is clear.

Proposition 2 *For an IOSTS* $\mathcal{S}$ *and* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$, $\mathcal{I}^\delta \not\models (\mathit{test}(\mathcal{S}, \omega), \mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}) \Rightarrow \neg(\mathcal{I}$ **ioco** $\mathcal{S})$.

Proposition 2 says that when $\mathit{test}(\mathcal{S}, \omega)$ enters a location in the set $\mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}$ (= $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_{\mathcal{S}^\delta}\}$) when executed on an implementation $\mathcal{I}$, then $\mathcal{I}$ violates conformance to $\mathcal{S}$. The property ω is not violated (the $\mathit{Violate}_\omega$ set is not entered). In this case, the ***Fail*** verdict is given:

Fail: the implementation violates the conformance, but not the property.

A third interpretation of the IOSTS $\omega||\Sigma^!(M)$ as an observer can be given, by choosing the set of violating locations to be $\mathit{Violate}_\omega \times Q_M$. We denote this set by $\mathit{Violate}_{\omega||\Sigma^!(M)}$, and omit the subscript whenever it is clear from the context.

Proposition 3 *For an IOSTS* $\mathcal{S}$ *and observer* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$, $\mathcal{I}^\delta \not\models (\mathit{test}(\mathcal{S}, \omega), \mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)}) \Rightarrow \mathcal{I}^\delta \not\models (\omega, \mathit{Violate}_\omega) \wedge \mathcal{S}^\delta \not\models (\omega, \mathit{Violate}_\omega)$.

Proposition 3 says that when $\mathit{test}(\mathcal{S}, \omega)$ enters a location in $\mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)}$ when executed on an implementation $\mathcal{I}$, then a violation of the property by both specification and implementation is detected. Hence, the given verdict:

Violate: the specification and the implementation violate the property.

Discussion. Propositions 1, 2, and 3 show that the test generation algorithm, i.e., the construction of the IOSTS $\mathit{test}(\mathcal{S}, \omega)$ and of its three verdicts, are *correct*, in the sense that verdicts correctly describe the relations between specification, implementation, and property. The verdict ***ViolateFail*** (resp. ***Fail***) detects the violation of the property and of the conformance (resp. of the conformance only) by the implementation. This holds independently of whether the specification satisfies the property or not; indeed, the execution of the test case on the implementation may detect violations of the property by the specification using the ***Violate*** verdict. The ability to generate test cases from a property and a specification which may or may not satisfy the property is important, because verification is undecidable for the infinite-state systems considered in this paper.

A natural question that arises is why a violation of the property by the implementation is always detected simultaneously with either (1) a violation of the property by the specification or (2) a violation of the conformance between implementation and specification. The reason is that our test cases are extracted from the specification, i.e., they only contain traces of the specification. An implementation may only violate a property without (1) or (2) occurring when it executes a trace that diverges at some point from the specification by an *input*; indeed, as seen in Section 3.4, this does not compromise conformance and,

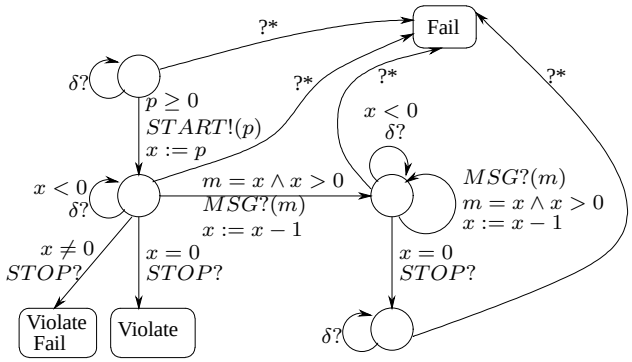


Figure 4. Before selection: test case obtained from $\mathcal{S}$ (Figure 1) and ω_2 (Figure 3).

of course, the specification cannot violate the property on a trace that it does not contain. Such traces are excluded from the generated test cases by construction.

Alternatively, these traces could be included in the test cases, but this implies to perform an *input-completion* of the specification (similar to Definition 12) first, and could lead to test cases that are typically too large for use in practice.

Building an actual test case. To build an actual test case from *test*$(\mathcal{S}, \omega)$, all inputs are transformed into outputs and reciprocally (this operation is called *mirror* ; in the test execution process, the actions of the implementation and those of the test case must complement each other). For the IOSTS $\mathcal{S}$ depicted in Figure 1 and the observer ω_2 depicted in Figure 3, the corresponding test case (before simplification) is depicted in Figure 4. Finally, the result is automatically analysed and simplified using the NBac tool [13] for statically eliminating transitions that cannot lead to the violation of the property any more (cf. Section 5).

5. Test selection

The main goal of the testing process is to detect violations of the system's required properties by the system's implementation. In this section we outline a technique for statically detecting and eliminating locations and transitions of a test case (generated from a specification and a property as described in Section 4) from which this goal cannot be achieved any more; the resulting test case attempts to keep the implementation in states where it may still violate the property. We show that this optimisation preserves correctness of test verdicts.

The violation of a property - described as an observer $(\omega, \mathit{Violate}_\omega)$ - by an implementation is materialised by reaching the *ViolateFail* and *Violate* sets of locations in the IOSTS *test*$(\mathcal{S}, \omega)$(cf. Section 4). For a state s of an IOSTS and a location q of the IOSTS, we say that s is *coreachable* for the location q if there exists a valuation v of the variables such that $s \xrightarrow{\sigma} \langle q, v \rangle$. Then, the test selection process consists (ideally) in selecting, from a given test case, the subset of states that are coreachable for the locations in *Violate* $\cup$ *ViolateFail*.

It should be quite clear that an exact computation of this set of states is impossible in general. However, there exist techniques that allow to compute an over-approximation of it. We here use one such technique based on abstract interpretation and implemented in the NBac tool [13]. Given a location q of an IOSTS, the tool computes, for each location l, a *symbolic coreachable state* for q:

Definition 14 (symbolic coreachable state) *For l, q two locations of an IOSTS $\mathcal{S}$, we say $\langle l, \varphi_{l \to q} \rangle$ is a* symbolic coreachable state *for q if $\varphi_{l \to q}$ is a formula on the variables of the IOSTS such that, if a state of the form $\langle l, v \rangle$ is coreachable for q, then $v \models \varphi_{l \to q}$ holds.*

I.e., $\langle l, \varphi_{l \to q} \rangle$ over-approximates the states with location l that are coreachable for q. The following algorithm uses this information for *pruning* a test case.

Definition 15 (pruning) *For an IOSTS $\mathcal{S}$ and an observer $(\omega, \mathit{Violate}_\omega)$ from the set $\Omega(\mathcal{S}^\delta)$, let prune$(\mathcal{S}, \omega)$ be the IOSTS computed as follows.*

- *first, the IOSTS $mirror(\mathit{test}(\mathcal{S}, \omega))$ is computed as in Section 4. Let L be its set of locations, $\mathcal{T}$ its set of transitions, and $\Sigma = \Sigma^! \cup \Sigma^?$ its alphabet, where $\Sigma^! = \Sigma^?_{\mathcal{S}}$ and $\Sigma^? = \Sigma^!_{\mathcal{S}} \cup \{\delta\}$. Let also Inconc $\notin L$ be a new location.*
- *then, for each location $l \in L$, a symbolic coreachable state $\langle l, \varphi_{l \to q} \rangle$, for each location $q \in$ Violate $\cup$ ViolateFail is computed. Let φ_l denote the formula $\bigvee_{q \in \mathit{Violate} \cup \mathit{ViolateFail}} \varphi_{l \to q}$*
- *next, for each location $l \in l$ of the IOSTS, and each transition $t \in \mathcal{T}$ of the IOSTS with origin l, guard G, and label a,*
  - *if $a \in \Sigma^!$ then*
    - *if $G \wedge \varphi_l$ is unsatisfiable, then t is eliminated from $\mathcal{T}$,*
    - *otherwise, the guard of t becomes $G \wedge \varphi_l$*
  - *if $a \in \Sigma^?$, then*
    - *the guard of t becomes $G \wedge \varphi_l$*
    - *a new transition is added to $\mathcal{T}$, with origin l, destination Inconc, action a, guard $G \wedge \neg\varphi_l$, and identity assignments.*

The *pruning* operation consists in detecting transitions whose firing leads to states where the *Violate* and *ViolateFail* sets of locations are unreachable. This is done by performing a coreachability analysis to these locations using the NBac tool [13]. If such a "useless" transition is labelled by an *output*, then it may be removed from the

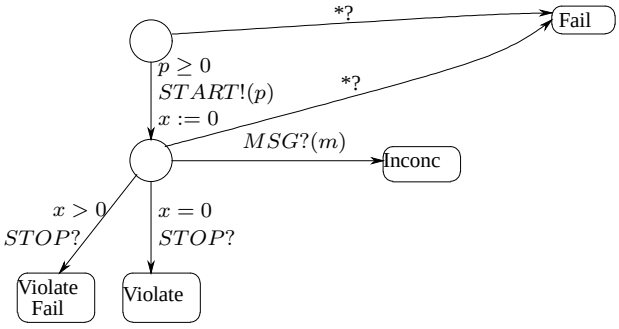


Figure 5. After selection: test case obtained from $\mathcal{S}$ (Figure 1) and ω_2 (Figure 3).

test case (a test case controls its outputs, hence, it may decide not to perform an output if violations of the property cannot be detected afterwards). On the other hand, *inputs* cannot be prevented from occurring, hence, the transitions labelled by *inputs*, by which the *Violate* and *ViolateFail* sets of locations cannot be reached any more, are reoriented to a new location, called *Inconc*. Reaching *Inconc* during test execution is interpreted as a verdict:

Inconc: violations of the property cannot be detected any more.

Proposition 4 *The test case obtained by after pruning is correct, i.e., Propositions 1, 2 and 3 still hold when test*$(\mathcal{S}, \omega)$ *is replaced with prune*$(\mathcal{S}, \omega)$.

The test case obtained after pruning *test*$(\mathcal{S}, \omega_2)$ is depicted in Figure 5. It starts by sending a *START* with a positive parameter p to the implementation, and then waits for inputs. If the implementation replies with *STOP*, the test execution terminates with a verdict, which depends on whether the parameter p was strictly positive or was equal to zero:

- If $p > 0$, the sequence $START(p) \cdot STOP$ exhibits a non-conformance between implementation (which accepts this sequence) and specification (which does not accept it). This sequence is also a witness for the violation of the property by the implementation: the verdict is *ViolateFail*;
- If $p = 0$, $START(p) \cdot STOP$ is a witness for violation of the property defined by ω_2 by both implementation and specification: the verdict is *Violate*.

Finally, if the implementation replies with *MSG* after *START*, the current test case cannot detect violations of the property any more, and the verdict is *Inconc*.

6. Conclusion and Related Work

A system may be viewed at several levels of abstraction: high-level *properties*, operational *specification*, and black-box *implementation*. In our framework properties and specifications are described using Input-Output Symbolic Transition Systems (IOSTS), which are extended automata that operate on symbolic variables and communicate with the environment through input and output actions carrying parameters. IOSTS are given a formal semantics in terms of input-output labelled transition systems (IOLTS). The implementation is a black box, but it is assumed that its semantics can be described by an unknown IOLTS. This allows to formally link the implementation and the specification by a conformance relation. A satisfaction relation links them both to higher-level properties.

A validation methodology is proposed for checking these relations, i.e., for detecting inconsistencies between the different views of the system: First, the properties are automatically verified on the specification using abstract interpretation techniques. Then, test cases are automatically generated from the specification and the properties, and are executed on the implementation of the system. If the verification step was successful, that is, it has established that the specification satisfies a property, the test execution may detect the violation of the property by the implementation and the violation of the conformance relation between implementation and specification. On the other hand, if the verification did not allow to prove a property, the test execution may additionally detect a violation of the property by the specification. Any inconsistencies obtained in this manner are reported to the user in the form of test verdicts. The approach is proved correct and is illustrated on a simple example. The full version of this paper [17] illustrates the approach on a larger example (the BRP protocol [11]).

Related Work. In [8] an approach for generating tests from a specification and from observers describing linear-time temporal logic requirements is described. The generated test cases do not check for conformance, they only check the fact that the implementation does not violate the requirements.

The approach described in [2] considers a specification S and an invariant P assumed to hold on S. Then, mutants S' of S are built using standard mutation operators, and a combined machine is generated, which extends sequences of S with sequences of S'. Next, a model checker is used to generate sequences that violate P, which prove that S' is a mutant of S violating P. Finally, the obtained sequences are interpreted as test cases to be executed on the implementation.

The authors of [9] start from a specification S and a temporal-logic property P assumed to hold on S, and use the ability of model checkers to construct counter-examples for $\neg P$ on S. These counter-examples can be interpreted as *witnesses* (i.e., test cases) for P on S. The papers [3, 12] extend this idea by formalising standard coverage criteria (all-definitions, all-uses, *etc*) using observers (resp. in temporal logic). Again, test cases are generated by model checking the observers (or the

temporal-logic formulas) on the specification.

The approaches described in all these papers rely on model checking, hence, they only work for finite-state systems; moreover, they do not formally relate satisfaction of properties to conformance testing, and, except for [8], they do not formally define a conformance relation.

In [18] we present an approach for combining model checking and conformance testing for finite-state systems, which can be seen as a first step of the approach presented here, which deals with infinite-state systems. In the finite-state framework of [18] verification is decidable, which heavily influences the whole approach: for example the test generation algorithm (based on enumerative model checking) does not need to take into account the possibility that the property might be violated by the specification.

A different approach for combining model checking and black-box testing is black-box checking [16]. Under some assumptions on the implementation (the implementation is deterministic; an upper bound n on its number of states is known), the black-box checking approach constructs a complete test suite of size exponential in n for checking properties expressed by Büchi automata.

Our approach can also be related to the combination of verification, testing and monitoring proposed in [10]. In their approach, monitoring is passive (pure observation), whereas ours is reactive and adaptative, guided by the choice of inputs to deliver to the system as pre-computed in a test case.

Finally, in [14] we propose a symbolic algorithm for selecting test cases from a specification be means of so-called *test purposes*. The difference with the present paper lies mainly in methodology. Test purposes in [14] are essentially a pragmatic means for test selection - they have to be provided by the user. In contrast, test selection in the present paper consists in automatically attempting to violate a safety property that was automatically verified (succesfully or not) on the specification. Moreover, test purposes can be classified as *reachability* properties, which have an exactly opposite semantics to the safety properties considered here (reachability properties are negations of safety properties).

References

[1] ISO/IEC 9646. Conformance Testing Methodology and Framework, 1992.

[2] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2001.

[3] J. Blom, A. Hessel, B. Jonnson, and P. Pettersson. Specifying and generating test cases using observer automata. In *Workshop on Formal Approaches to Software Testing (Fates'04)*, pages 137–152, 2004.

[4] E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification (PSTV'88)*, pages 63–74, 1988.

[5] E. Brinskma, A. Alderen, R. Langerak, J. van de Laagemat, and J. Tretmans. A formal approach to conformance testing. In *Protocol Secification, Testing and Verification (PSTV'90)*, pages 349–363, 1990.

[6] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, number 2280 in LNCS, pages 470–475, 2002.

[7] P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[8] J.C. Fernandez, L. Mounier, and C. Pachon. Property-oriented test generation. In *Formal Aspects of Software Testing Workshop*, number 2931 in LNCS, 2003.

[9] A. Gargantini and C.L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/SIGSOFT FSE*, pages 146–162, 1999.

[10] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Int. Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France*, number 2280 in LNCS, pages 342–356, 2002.

[11] L. Helmink, M. P. A. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In *Types for Proofs and Programs (TYPES'94)*, number 806 in LNCS, pages 127–165, 1994.

[12] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, number 2280 in LNCS, pages 327–341, 2002.

[13] B. Jeannet. Dynamic partitioning in linear relation analysis. *Formal Methods in System Design*, 23(1):5–37, 2003.

[14] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *Int. Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05), Grenoble, France (to appear)*, 2005.

[15] N. Lynch and M. Tuttle. Introduction to IO automata. *CWI Quarterly*, 3(2), 1999.

[16] D. Peled, M. Vardi, and M. Yannakakis. Black-box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225 – 246, 2001 2001.

[17] V. Rusu, H. Marchand, and T. Jéron. Verification and symbolic test generation for safety properties. Technical Report 1640, IRISA, august 2004. Available at `http://www.irisa.fr/vertecs/Publis/Ps/PI-1640.pdf`.

[18] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verifcation to safety testing. In *Intl. Conf. on Testing of Communicating Systems (TestCom04)*, number 2978 in LNCS, 2004.

[19] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, number 1664 in LNCS, pages 46–65, 1999.

[20] E. Zinovieva. *Symbolic Test Generation for Reactive Systems*. PhD thesis, University of Rennes I, November 2004.

An Introduction to Timed Automata

Patricia Bouyer
LSV – CNRS & ENS de Cachan
61, avenue du Président Wilson
94230 Cachan – France
email: bouyer@lsv.ens-cachan.fr

1 Introduction

Explicit timing constraints are naturally present in real-life systems (transmission delays, response time, etc...). Classical models (finite automata, Petri nets, etc...) can not express such real-time constraints. Since their introduction by Rajeev Alur and David Dill in [6, 7], timed automata are one of the most studied models for real-time systems: in those systems, quantitative properties of delays between events can easily be expressed. Numerous works have been devoted to the "theoretical" comprehension of timed automata: determinization [9], minimization [3], power of clocks [5, 33], power of ε-transitions [15], extensions of the model [27, 35, 23, 13], logical characterizations [35], etc... have in particular been investigated. Practical aspects of the model have also been considered and several model-checkers are now available (HYTECH [31], KRONOS [25], UPPAAL [38]). These model-checkers have been used to verify many industrial case studies (see the web pages of the tools, given page 13).

One of the major properties of timed automata is probably that reachability properties are decidable [7], though timed automata have an infinite number of configurations. The core of this result is the construction of the so-called region automaton, which finitely abstract behaviours of timed automata in such a way that checking reachability in a timed automaton reduces to checking reachability in a (somewhat larger) finite automaton. This construction has many other applications, as for example the decidability of the TCTL model-checking [2] (TCTL is the timed extension of the logic CTL). However, many problems remain undecidable, as not everything can be reduced to the untimed framework. For example, timed automata are neither determinizable, nor complementable [7]. Checking if a timed automaton is determinizable (or complementable) is even an undecidable problem [42]. An other important example is the undecidability of the universality problem for timed automata [7].

The aim of this tutorial is to give some understanding of the timed automata model. We will present the basic tools which are used in the domain of verification of timed systems. In particular, after having presented the model, we will present in details the region automata construction. For modeling reasons, it is important to have expressive models, but it is also important that the models remain decidable. We will then present several variants or extensions of timed automata, focusing on the decidability of reachability properties, and on the expressiveness of the models. We will terminate this tutorial with some implementation and algorithmics issues.

We would like to point out several recent surveys on timed automata which present current works and results on timed automata with a point of view somewhat different from the one adopted in this tutorial. A recent survey by Rajeev Alur and Madhusudan P. gives many hints about decidability issues for timed automata [10]. In [11], Eugene Asarin presents the current challenges in timed languages theory.

2 Timed Automata

If Z is a set, let Z^* be the set of *finite* sequences of elements in Z. We consider as time domain $\mathbb{T}$ the set $\mathbb{Q}_+$ of non-negative rationals or the set $\mathbb{R}_+$ of non-negative reals, and Σ as a finite set of *actions*. A *time sequence* over $\mathbb{T}$ is a finite non decreasing sequence $\tau = (t_i)_{1\leq i\leq p} \in \mathbb{T}^*$. A *timed word* $\omega = (a_i, t_i)_{1\leq i\leq p}$ is an element of $(\Sigma \times \mathbb{T})^*$, also

written as a pair $\omega = (\sigma, \tau)$, where $\sigma = (a_i)_{1 \leq i \leq p}$ is a word in Σ^* and $\tau = (t_i)_{1 \leq i \leq p}$ a time sequence in $\mathbb{T}^*$ of same length.

Clock Valuations, Operations on Clocks. We consider a finite set X of variables, called *clocks*. A *clock valuation* over X is a mapping $v : X \to \mathbb{T}$ which assigns to each clock a time value. The set of all clock valuations over X is denoted $\mathbb{T}^X$. Let $t \in \mathbb{T}$, the valuation $v + t$ is defined by $(v + t)(x) = v(x) + t$, $\forall x \in X$. We also use the notation $(\alpha_i)_{1 \leq i \leq n}$ for the valuation v such that $v(x_i) = \alpha_i$. For a subset Y of X, we denote by $[Y \leftarrow 0]v$ the valuation such that for each $x \in Y$, $([Y \leftarrow 0]v)(x) = 0$ and for each $x \in X \setminus Y$, $([Y \leftarrow 0]v)(x) = v(x)$.

Clock Constraints. Given a finite set of clocks X, we introduce two sets of *clock constraints over* X. The most general one, denoted $\mathcal{C}(X)$, is defined by the grammar:

$g ::= x \bowtie c \mid x - y \bowtie c \mid g \wedge g \mid \mathit{true}$
where $x, y \in X$, $c \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

We also use the proper subset of *diagonal-free* constraints where the comparison between two clocks is not allowed. This set, denoted $\mathcal{C}_{df}(X)$, is defined by the grammar:

$g ::= x \bowtie c \mid g \wedge g \mid \mathit{true}$,
where $x \in X$, $c \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A *k-bounded clock constraint* is a clock constraint which involves only constants c between $-k$ and $+k$. The set of k-bounded (resp. k-bounded diagonal-free) clock constraints is denoted $\mathcal{C}^k(X)$ (resp. $\mathcal{C}^k_{df}(X)$). A constraint of the form $x - y \bowtie c$ is a *diagonal constraint*.

If v is a clock valuation we write $v \models g$ when v satisfies the clock constraint g and we say that v satisfies $x \bowtie c$ (resp. $x - y \bowtie c$) whenever $v(x) \bowtie c$ (resp. $v(x) - v(y) \bowtie c$). If g is a clock constraint, we note $[\![g]\!]$ the set of clock valuations $\{v \in \mathbb{T}^X \mid v \models g\}$.

Timed Automata. A *timed automaton* over $\mathbb{T}$ is a tuple $\mathcal{A} = (\Sigma, Q, T, I, F, X)$, where Σ is a finite alphabet of actions, Q is a finite set of states, X is a finite set of clocks, $T \subseteq Q \times [\mathcal{C}(X) \times \Sigma \times 2^X] \times Q$ is a finite set of transitions[1], $I \subseteq Q$ is the subset of initial states and $F \subseteq Q$ is the subset of final states. If all constraints appearing in $\mathcal{A}$ are diagonal-free, we say that $\mathcal{A}$ is a *diagonal-free timed automaton.*

A *path* in $\mathcal{A}$ is a finite sequence of consecutive transitions:

$$P = q_0 \xrightarrow{g_1, a_1, Y_1} q_1 \ \ldots \ q_{p-1} \xrightarrow{g_p, a_p, Y_p} q_p$$

where $q_{i-1} \xrightarrow{g_i, a_i, Y_i} q_i \in T$ for every $1 \leq i \leq p$.

The path is said to be *accepting* if it starts in an initial state ($q_0 \in I$) and ends in a final state ($q_p \in F$). A *run* of the automaton along the path P is a sequence of the form:

$$(q_0, v_0) \xrightarrow[t_1]{g_1, a_1, Y_1} (q_1, v_1) \ \ldots \ \xrightarrow[t_p]{g_p, a_p, Y_p} (q_p, v_p)$$

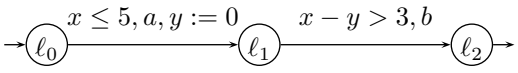
where $\tau = (t_i)_{1 \leq i \leq p}$ is a time sequence and $(v_i)_{1 \leq i \leq p}$ are clock valuations such that:

$$\begin{cases} v_0(x) = 0, \ \forall x \in X \\ v_{i-1} + (t_i - t_{i-1}) \models g_i \\ v_i = [C_i \leftarrow 0]\,(v_{i-1} + (t_i - t_{i-1})) \end{cases}$$

The label of the run is the timed word $w = (a_1, t_1) \ldots (a_p, t_p)$. If the path P is accepting then the timed word w is said to be accepted by $\mathcal{A}$. The set of all timed words accepted by $\mathcal{A}$ is denoted by $L_t(\mathcal{A})$.

Remark 1 *In these notes, we only consider finite paths and words with finitely many actions, but we could consider more general acceptance conditions (Büchi, Muller, etc...) as well, see [7].*

Example 1 *An example of timed automaton is given below.*



This timed automaton accepts the timed word $(a, 4.1)(b, 5.5)$. *An accepting run for this word is*

$$(\ell_0, (0, 0)) \xrightarrow[4.1]{a} (\ell_1, (4.1, 0)) \xrightarrow[5.5]{b} (\ell_2, (5.5, 1.4))$$

where $(4.1, 0)$ *represents the valuation* v *such that* $v(x) = 4.1$ *and* $v(y) = 0$.

3 Reachability Analysis

For verification purposes, the most fundamental properties that one should be able to verify are reachability properties: safety properties can for

[1] For more readability, a transition will often be written as $q \xrightarrow{g, a, Y} q'$ or even as $q \xrightarrow{g, a, Y := 0} q'$ instead of simply the tuple (q, g, a, Y, q').

example be expressed as reachability properties. Usually a class of models is said *decidable* whenever checking reachability properties in this class is decidable. Otherwise this class is said *undecidable*. For timed automata reachability properties we want to check are: "Is state q of timed automaton $\mathcal{A}$ reachable? *i.e.* is there a run starting in an initial state leading to q?" There is no requirement as what are the values of the clocks when reaching state q. This problem is equivalent to the *emptiness problem* (from a language-theoretical point of view), where the question is whether the language accepted by a timed automaton is empty or not.

The class of finite automata is obviously decidable, the reachability problem is even NLOGSPACE-complete [36], and efficient methods, symbolic techniques, data structures, etc... have been developed and implemented [24]. The problem with timed automata is that the number of configurations of a timed automaton is infinite (a configuration is a pair (q, v) where q is a state and v a clock valuation). Techniques used for verifying finite automata can thus not be used for timed automata. Specific symbolic techniques and abstractions have to be developed, which take into account the specific properties of timed automata, in particular the fact that clocks evolve synchronously with global time.

In the following, we will concentrate on the verification of reachability properties in timed automata, and present the basic technics for solving this problem. Of course, in the literature, more general properties have been considered. For example, the model-checking of TCTL [2], a timed extension of CTL, is decidable in PSPACE, and symbolic technics have been developed to efficiently model-check TCTL [34]. Note however that not everything can be reduced to the finite untimed case using the region automaton construction: for example, universality of timed automata is undecidable [7], and model-checking of most linear-time timed temporal logics are undecidable, when equality can be used in the constraints [8].

4 The Region Abstraction

The construction we will describe below is due to Alur and Dill first in [6]. The aim of this construction is to finitely abstract behaviours of timed automata, so that checking a reachability property in a timed automaton reduces to checking a reachability property in a finite automaton.

4.1 The Region Automaton Construction

Region Partitioning. Let us fix a finite set of clocks X. Let $\mathcal{R}$ be a finite partitioning of $\mathbb{T}^X$. Let $\mathcal{C}$ be a finite set of constraints over X. We define three compatibility conditions as follows:

① We say that $\mathcal{R}$ is *compatible with constraints* $\mathcal{C}$ if for every constraint g in $\mathcal{C}$, for every R in $\mathcal{R}$, either $[\![g]\!] \subseteq R$ or $[\![g]\!] \cap R = \emptyset$.

② We say that $\mathcal{R}$ is *compatible with elapsing of time* if for all R and R' in $\mathcal{R}$, if there exists some $v \in R$ and $t \in \mathbb{T}$ such that $v + t \in R'$, then for every $v' \in R$, there exists some $t' \in \mathbb{T}$ such that $v' + t' \in R'$.

③ We say that $\mathcal{R}$ is *compatible with resets* whenever for all R and R' in $\mathcal{R}$, for every subset $Y \subseteq X$, if $[Y \leftarrow 0]R \cap R' \neq \emptyset$, then $[Y \leftarrow 0]R \subseteq R'$.

If $\mathcal{R}$ satisfies these three conditions, we will say that $\mathcal{R}$ is a *set of regions* for the set of constraints $\mathcal{C}$ or simply a set of regions (if $\mathcal{C}$ is clear from the context). $\mathcal{R}$ defines in a natural way an equivalence relation $\equiv_{\mathcal{R}}$ over valuations ($v \equiv_{\mathcal{R}} v'$ iff for each region R of $\mathcal{R}$, $v \in R \iff v' \in R$). An equivalence class of $\equiv_{\mathcal{R}}$ (or equivalently an element of $\mathcal{R}$) is called a *region*. If v is a valuation we note $[v]_{\mathcal{R}}$ the region to which v belongs.

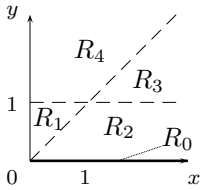
The intuition behind these conditions is the following: we want to finitely abstract behaviours of timed automata. To this aim, we finitely abstract the (infinite) set of valuations: a valuation v will be abstracted by the region $[v]_{\mathcal{R}}$. In order for the abstraction to preserve (at least) reachability properties, it must be the case that if two valuations are equivalent, then their future behaviours are also equivalent. The three conditions above precisely express this property: condition ① says that two equivalent valuations satisfy the same clock constraints, condition ② says that elapsing of time does not distinguish two equivalent valuations whereas condition ③ says that resetting clocks does not distinguish two equivalent valuations.

Region Graph. From a set of regions $\mathcal{R}$ one can define the so-called *region graph*, which represents the possible timing evolutions of the system: the region graph is a finite automaton whose set of states is $\mathcal{R}$ and whose transitions are:

$$\begin{cases} R \xrightarrow{\varepsilon} R' \text{ if } R' \text{ is a time successor of } R \\ R \xrightarrow{Y} R' \text{ if } [Y \leftarrow 0]R \subseteq R' \end{cases}$$

Intuitively, the region graph records possible timed evolutions of the system: there is a transition $R \xrightarrow{\varepsilon} R'$ if, from every valuation of R, it is possible to let some time elapse and reach R'. There is a transition $R \xrightarrow{Y} R'$ if, from R, R' can be reached by resetting clocks in Y.

Example 2 *Let us consider the following partitioning of* $\mathbb{R}_+^{\{x,y\}}$.



It is easy to verify that $\mathcal{R}$ *is a set of regions for the constraints* $\{y = 1, x = y\}$. *The region graph associated with* $\mathcal{R}$ *is represented on* **Fig.** *1.*

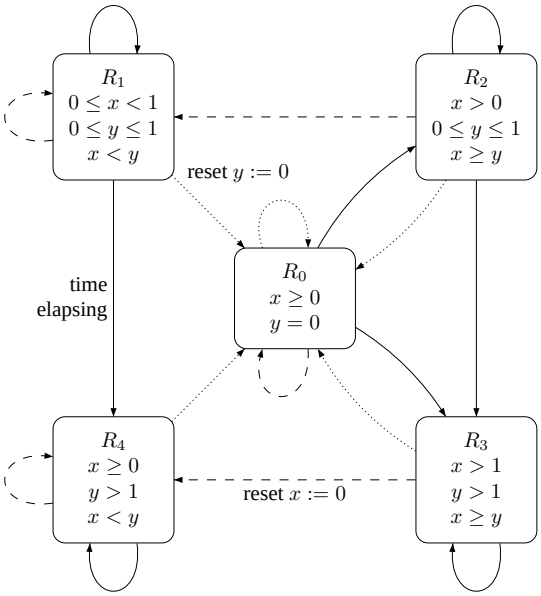


Fig. 1. A simple example of region graph

Region Automaton. Consider a timed automaton $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ with set of constraints $\mathcal{C}$. Let $\mathcal{R}$ be a finite set of regions for $\mathcal{C}$ (*i.e.* a partitioning of $\mathbb{T}^X$ satisfying conditions ①, ② and ③). The *region automaton* $\Gamma_{\mathcal{R}}(\mathcal{A})$ is the finite automaton whose set of states is $Q \times \mathcal{R}$, whose initial states are $I \times \{R_0\}$ (where R_0 is the region containing the valuation assigning 0 to each clock), whose final states are $F \times \mathcal{R}$ and whose transitions are defined as follows:

- there is a transition $(\ell, R) \xrightarrow{a} (\ell', R')$ whenever there exists a transition $\ell \xrightarrow{g,a,Y} \ell'$ in $\mathcal{A}$ with $R \subseteq [\![g]\!]$ and $R \xrightarrow{Y} R'$ transition of the region graph
- there is a transition $(\ell, R) \xrightarrow{\varepsilon} (\ell, R')$ whenever $R \xrightarrow{\varepsilon} R'$ transition of the region graph

This automaton somehow simulates the original timed automaton: the first type of transitions simulates discrete actions (or transitions) whereas the second type of transitions simulates elapsing of time.

The fundamental property of this construction is the following:

Proposition 1 *Let* $\mathcal{A}$ *be a timed automaton with set of constraints* $\mathcal{C}$. *We assume we can construct a set of regions* $\mathcal{R}$ *for* $\mathcal{C}$. *Then,*

$$\mathsf{Untime}(L_t(\mathcal{A})) = L(\Gamma_{\mathcal{R}}(\mathcal{A}))$$

where $L(\Gamma_{\mathcal{R}}(\mathcal{A}))$ *is the (untimed) language accepted by* $\Gamma_{\mathcal{R}}(\mathcal{A})$, *and*

$$\mathsf{Untime}((a_1, t_1) \dots (a_p, t_p)) = a_1 \dots a_p.$$

More precisely, whenever in $\mathcal{A}$ we can wait some delay and do an a, then in $\Gamma_{\mathcal{R}}(\mathcal{A})$, we can take several ε-transitions and then do an a, and *vice-versa*. We will see in section 4.3 that this property naturally expresses in terms of time-abstract bisimulation. Checking reachability properties in $\mathcal{A}$ thus reduces to checking reachability properties in $\Gamma_{\mathcal{R}}(\mathcal{A})$. As $\Gamma_{\mathcal{R}}(\mathcal{A})$ is a finite automaton, we get that for every timed automaton $\mathcal{A}$ for which we can construct a set of regions (satisfying conditions ①, ② and ③), we can decide reachability properties using the region automaton construction

4.2 Region Automaton for Classical Timed Automata

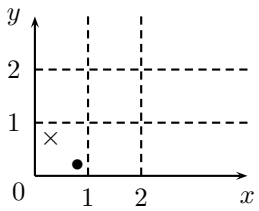
We fix for this subsection a finite set of clocks X.

Sets of regions for diagonal-free constraints. Let M be an integer. We define the following partitioning of $\mathbb{T}^X$. Let v and v' be two valuations of $\mathbb{T}^X$, we say that $v \equiv^M_{df} v'$ if all three following conditions hold:

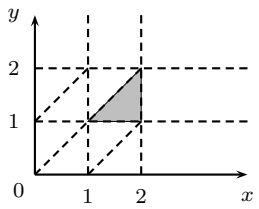
- $v(x) > M$ iff $v'(x) > M$ for each $x \in X$,
- if $v(x) \le M$, then $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ and $\Big(\{v(x)\} = 0$ iff $\{v'(x)\} = 0\Big)$ for each $x \in X$, and

- if $v(x) \leq M$ and $v(y) \leq M$, then $\{v(x)\} \leq \{v(y)\}$ iff $\{v'(x)\} \leq \{v'(y)\}$ for all $x,\ y \in X$.

The relation $\equiv_{df}^{M}$ is an equivalence relation of finite index. The partitioning $\mathcal{R}_{df}^{M}(X)$ is then defined as the set of equivalence classes of $\mathbb{T}^{X}/_{\equiv_{df}^{M}}$. **Fig.** 2 explains the region construction for two clocks.



(a) Partition compatible with constraints, not with time elapsing (the two points • and × can not be equivalent)



region defined by:

$$\begin{cases} 1 < x < 2 \\ 1 < y < 2 \\ \{x\} < \{y\} \end{cases}$$

(b) Partition compatible with constraints, time elapsing (and resets)

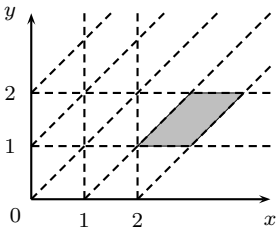
Fig. 2. Diagonal-free region partitioning for two clocks and maximal constant 2

It is easy to prove (and left as an exercise) the following lemma:

Lemma 1 *The partitioning $\mathcal{R}_{df}^{M}(X)$ is a set of regions for the constraints $\mathcal{C}_{df}^{M}(X)$.*

Roughly counting all possible combinations above, we can bound the number of regions in $\mathcal{R}_{df}^{M}(X)$ by $2^{|X|}.|X|!.(2M+2)^{|X|}$ where $|X|$ is the cardinal of X.

Sets of regions for general constraints. Recall that the difference between diagonal-free clock constraints and general clock constraints stands in the fact that *diagonal constraints* (*i.e.* constraints of the form $x - y \bowtie c$) can be used. An easy extension of the previous construction can be done. We do not define it formally here, but only give a simple example with two clocks, see **Fig.** 3.



region defined by:

$$\begin{cases} 2 < x \\ 1 < y < 2 \\ 1 < x - y < 2 \end{cases}$$

Fig. 3. Set of regions for 2**-bounded general constraints with two clocks**

This set of regions is denoted $\mathcal{R}^{M}(X)$, and its cardinal can roughly be bounded by $(2M+2)^{(|X|+1)^2}$. Note that this set of regions is also correct for M-bounded diagonal-free constraints.

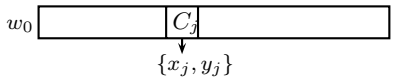
Region automata for classical timed automata. Let $\mathcal{A}$ be a timed automaton with set of clocks X. Let M be the maximal constant involved in one of the constraints of $\mathcal{A}$, the set $\mathcal{R}^{M}(X)$ is a set of regions for $\mathcal{A}$. From the results of the previous subsections, we get the following theorem, due to Alur and Dill [6, 7], which is the core of the verification of timed systems.

Theorem 1 (Alur & Dill 90's) *Reachability (or equivalently emptiness) is decidable for timed automata. It is a* PSPACE*-complete problem (for both diagonal-free as well as general timed automata).*

Although this theorem has been first proved in [7], the proof we choose to sketch is taken from [1], where it is written in details.

Proof. [Sketch] PSPACE membership is easy: the size of th region automaton is exponential in the size of the original automaton. Using the NLOGSPACE complexity of the reachability problem in classical untimed graphs, we get that reachability in timed automata can be done in PSPACE.

PSPACE-hardness can be proved by reducing the termination of a linearly bounded Turing machine (LBTM for short) on some input to reachability in timed automata. The encoding is done as follows: assuming the alphabet is $\{a, b\}$, the content of cell C_j of the track of the LBTM is encoded by two clocks x_j and y_j. Cell C_j contains an "a" when the constraint $x_j = y_j$ holds, and cell C_j contains a "b" when the constraint $x_j < y_j$ holds. Note that these two conditions are invariant by time elapsing.



If $q \xrightarrow{\alpha,\alpha',\delta} q'$ is a transition of the LBTM, then for each position i of the tape, there will be a transition $(q,i) \xrightarrow{g,Y:=0} (q',i')$ where:

- g is $x_i = y_i$ (resp. $x_i < y_i$) if $\alpha = a$ (resp. $\alpha = b$)
- $Y = \{x_i, y_i\}$ (resp. $Y = \{x_i\}$) if $\alpha = a$ (resp. $\alpha = b$)
- $i' = i + 1$ (resp. $i' = i - 1$) if δ is right and $i < n$ (resp. left)

We need to enforce time elapsing; this can be done by adding a clock t which is checked to 1 and reset to 0 on all transitions. Initially the track contains the encoding of the word w_0. This can be done by a transition from a state "init" to $(q_0, 1)$ where q_0 is the initial state of the LBTM, which checks whether $t = 1$, and resets clocks in Y_0 where $Y_0 = \{t\} \cup \{x_i \mid w_0[i] = b\}$. The computation over w_0 of the LBTM terminates iff there is a run from state "init" to some state (q_f, i) where q_f is the final state of the LBTM. □

Note that the above encoding uses diagonal constraints, but as will be seen later (see section 5.1), there is no need of these diagonals. A direct but more involved construction without diagonals can be found in the appendix of [1].

Remark 2 *Note that sets of regions we have described could be refined: there is no need to have the same maximal constant for all clocks, one maximal constant for each clock could be used. However, for our purpose here, there is no need for such a refinement.*

4.3 Interpretation in Terms of Finite Bisimulation

With what has been presented before, conditions ①, ② and ③ (compatibility of the set of regions with constraints, time elapsing and resets) have a natural interpretation in terms of ***time-abstract bisimulation***.

Timed transition system associated with a timed automaton. We have defined the semantics of a timed automaton as runs or timed words. We could have defined its semantics as a timed transition system as well. Transition systems (thus in particular timed transition systems) are more suitable for behavioural comparisons of systems.

Let $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ be a timed automaton. The timed transition system associated with $\mathcal{A}$ has $Q \times \mathbb{T}^X$ for set of states and its transition relation is defined by the two following rules:

$$\begin{cases} (\ell, v) \xrightarrow{d} (\ell, v+d) & \text{for every } d \in \mathbb{T} \\ (\ell, v) \xrightarrow{a} (\ell', v') & \text{if there is } \ell \xrightarrow{g,a,Y} \ell' \text{ s.t. } v \models g,\ v' = [Y \leftarrow 0]v \end{cases}$$

Time-abstract bisimulation. Time-abstract bisimulation could be defined for two timed automata, but for our purpose, we follow the lines of [22] and define time-abstract bisimulation on a single timed automaton. Let $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ be a timed automaton (over alphabet Σ). We say that a relation $\equiv \subseteq (Q \times \mathbb{T}^X) \times (Q \times \mathbb{T}^X)$ is a *time-abstract bisimulation* whenever it is an equivalence relation satisfying the following conditions:

- if $(\ell_1, v_1) \equiv (\ell_2, v_2)$ and $(\ell_1, v_1) \xrightarrow{d_1} (\ell_1, v_1 + d_1)$ for some $d_1 \in \mathbb{T}$, then there exists $d_2 \in \mathbb{T}$ such that $(\ell_2, v_2) \xrightarrow{d_2} (\ell_2, v_2 + d_2)$ and $(\ell_1, v_1 + d_1) \equiv (\ell_2, v_2 + d_2)$
- if $(\ell_1, v_1) \equiv (\ell_2, v_2)$ and $(\ell_1, v_1) \xrightarrow{a} (\ell_1', v_1')$, then there exists (ℓ_2', v_2') such that $(\ell_2, v_2) \xrightarrow{a} (\ell_2', v_2')$ and $(\ell_1', v_1') \equiv (\ell_2', v_2')$
- and *vice-versa*.

By definition, such a relation is an equivalence relation, and as such, $\equiv$ is said to have a *finite index* whenever there are finitely many equivalence classes. Informally, from two equivalent configurations, it is possible to do the same discrete actions and/or to wait some amount of time (possibly different in the two configurations) and stay equivalent.

Relation with the region automaton construction.

Proposition 2 *Let $\mathcal{A}$ be a timed automaton and $\mathcal{R}$ a set of regions for the constraints in $\mathcal{A}$. The relation $\{((\ell, v), (\ell, v')) \mid [v]_{\mathcal{R}} = [v']_{\mathcal{R}}\}$ is a time-abstract bisimulation with a finite index.*

Time-abstract bisimulation appears indeed as the right notion corresponding to the region automaton construction and formally justifies everything which has been explained previously. It proves more precisely that the region automaton construction can be used to verify all properties that are

invariant by time-abstract bisimulation, *e.g.* reachability properties, safety properties, many untimed properties. However, notice that we can not use directly this construction to verify properties expressed in a timed logic like TCTL because a property like "reaching a state in exactly 5 units of time" is not invariant by time-abstract bisimulation. For these properties a more involved construction is needed which adds a clock for the formula, and then construct a region automaton taking into account this additional clock. We do not develop this construction here but better refer to original articles on the subject [2].

The converse of Proposition 2 also holds and it can be used to prove decidability of timed systems: if for a timed system we can compute a time-abstract bisimulation relation with a finite index, then reachability (and other time-abstract invariant properties) can be decided using a region automaton-like construction. Examples of such constructions can for example be found in [29, 22].

4.4 Partial Conclusion

Timed automata are an interesting model for representing systems with real-time constraints. Despite the infinite number of possible configurations of a timed automaton, model-checking of reachability properties has been proved decidable. This is probably the most fundamental property of timed automata, which has been proved at the beginning of the 90's by Alur and Dill, and which is the starting point of numerous works on timed models. We have presented in this section the basics of the decidability of timed automata, which relies on a reduction to finite automata: this is fundamental for most of the works on timed systems. It is however worth to notice that not everything can be reduced to the finite automata case. For example (see [7] and also [42]),

- universality (the dual of reachability) is an undecidable problem;
- the class of timed languages accepted by timed automata is not closed under complementation;
- not all timed automata can be determinized, and, in addition, the problem of deciding whether a timed automaton can be determinized is an undecidable problem;

These problems will not be tackled in this tutorial, but we refer to [10] for a survey of (un)decidability results about timed automata.

In the rest of this tutorial, we will mostly consider extensions (or variants) of timed automata and study decidability of these models, and we will also concentrate on algorithmics and implementation aspects. We hope this should help better understanding timed behaviours and timed models.

5 Extensions of Timed Automata

For representing real-life systems, it is much convenient to have expressive and easy-to-use models. We will present in this section several extensions (or variants) of timed automata, and will focus on the decidability of their reachability problem. We will also give some expressiveness results.

A class of systems $\mathcal{S}$ is said *strictly more expressive* than a class of systems $\mathcal{S}'$ whenever there exists S in $\mathcal{S}$ such that no S' in $\mathcal{S}'$ accepts the same language as S, and for every system S' in $\mathcal{S}'$, there exists S in $\mathcal{S}$ which recognizes the same language as S'. A class of systems $\mathcal{S}$ is *as expressive as* $\mathcal{S}'$ whenever for every S in $\mathcal{S}$, there exists S' in $\mathcal{S}'$ which accepts the same language as S.

5.1 Role of Diagonal Clock Constraints

Diagonal constraints (*i.e.* clock constraints of the form $x - y \bowtie c$ where $x,\ y \in X$, $c \in \mathbb{Z}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$) have been first mentioned in the seminal paper of Alur & Dill [7], and are often considered as part of the model of timed automata. We have seen in previous section that diagonal constraints do not add any decidability and complexity problems to the model.

It was known as a folklore result that diagonal constraints can be eliminated from timed automata, and thus that they do not add expressive power to timed automata. A formal proof of this result has been done in [15].

Proposition 3 *For every timed automaton $\mathcal{A}$, possibly with diagonal constraints, there exists a timed automaton $\mathcal{B}$, with only diagonal-free constraints, which recognizes the same language. Note that $\mathcal{B}$ is* ***strongly bisimilar***[2] *to $\mathcal{A}$.*

This construction leads to an exponential (in the number of diagonal constraints) blowup of the number of states of the automaton, and this blowup is unavoidable as timed automata with diagonal

[2] Which means they are bisimilar (in a classical way) for actions taken in $\Sigma \cup \mathbb{T}$: if a system can do action, then so can also the other system, and if a system can wait d units of time, then so can also the other system.

constraints are exponentially more succinct than diagonal-free timed automata [19].

5.2 Adding Silent Actions

For finite automata, it is well-known that *silent actions* (also known as ε-*transitions* or *internal actions*) do not add expressive power to finite automata and that they can be eliminated with no blowup in the number of states of the automaton. Silent actions in timed automata have been studied in details in [15], and the situation is far from the one in the untimed framework.

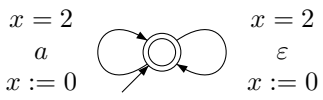
A first (easy) fact is that the region automaton construction can be done in a similar way when there are silent actions, we thus get:

Proposition 4 *The reachability problem is decidable for timed automata with silent actions. The complexity is also* PSPACE*-complete.*

However, and this is at first surprising, silent actions can not be removed, as it is the case for classical finite automata.

Theorem 2 *Timed automata with silent actions are strictly more expressive than classical timed automata.*

Several examples are given in [15]. Among them, there is the language $L = \{(a,t_1)\dots(a,t_i)\cdots \mid \forall i,\ i \mod 2 = 0\}$. This timed language is recognized by the following automaton but is recognized by no timed automaton without silent actions.



Proofs of non-expressivity by a classical timed automaton are always *ad-hoc* as there is no real criterion for a timed language to be recognized by a classical timed automaton. However a sufficient criterium is given in [15]: let $\mathcal{A}$ be a timed automaton possibly with silent actions; if, in $\mathcal{A}$, there is no loop in which a clock is reset on an ε-transition, then ε-transitions can be removed from $\mathcal{A}$, and we can construct a timed automaton $\mathcal{B}$ without ε-transitions which recognizes the same language.

5.3 Adding Additive Clock Constraints

We have seen that diagonal constraints can be used safely in timed automata. A natural idea is then to consider clock constraints of the form $x + y \bowtie c$. Such a constraint will be called an *additive clock constraint*. The model of timed automata which uses classical constraints and additive clock constraints has been studied in [16].

Two clocks. For timed automata with **two** clocks, a region construction can be done. We will not define it precisely here but the region partitioning when the maximal constant is 2 is illustrated on **Fig.** 4. The general case can be easily deduced from this representation.

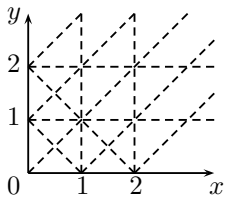
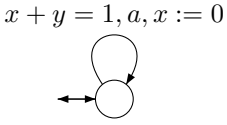


Fig. 4. Region partitioning for additive clock constraints (two clocks)

Proposition 5 *The reachability problem for timed automata with at most two clocks and possibly additive clock constraints is decidable.*

The language L^+ represented on **Fig.** 5 is accepted by a timed automaton with two clocks and additive clock constraints but is accepted by no timed automaton with classical clock constraints.



$L^+ = \{(a^n, t_1 \dots t_n) \mid n \geq 1 \text{ and } t_i = 1 - \frac{1}{2^i}\}$

Fig. 5. A language which needs additive clock constraints

Four clocks or more. The following result holds for timed automata with four clocks or more, and additive clock constraints:

Theorem 3 *The reachability problem is undecidable for timed automata with four clocks or more, and additive clock constraints.*

This undecidability result is rather involved and is by reduction from the halting problem of a two counter machine [39]. The proof can be found in [16].

What about three clocks? The region graph construction done for two clocks does not extend to three clocks. Using the characterization of regions using time-abstract bisimulation, it has been proven in [41] that there is no finite partitioning satisfying the conditions ①, ② and ③ as soon as there are three clocks (x, y and z) and constraints $\{x+y=1, x=0, z=1\}$ are used. However the reduction presented above (for proving undecidability of reachability checking in timed automata with four clocks and additive clock constraints) can not be adapted if we allow only three clocks. It is still an open problem to know if the reachability problem for timed automata with three clocks and additive clock constraints is decidable or not.

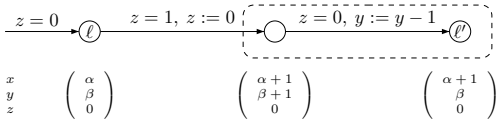
5.4 Adding New Operations on Clocks

Up to now, we can only reset clocks to zero. In [20], models using more general *updates* have been studied. In the model of *updatable timed automata*, a transition is of the form $\ell \xrightarrow{g,a,\mathsf{up}} \ell'$ where g is a clock constraint, a is an action and up is an *update, i.e.* for each clock x, an operation up_x of the form $x :\bowtie c$ or $x :\bowtie y + c$ where $c \in \mathbb{Z}$, y is a clock, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let us take two valuations v and v'. We have that $v' \in \mathsf{up}(v)$ whenever for each clock x, $v'(x) \in \mathsf{up}_x(v)$, where $\mathsf{up}_x(v) = \begin{cases} \{\alpha \mid \alpha \bowtie c\} & \text{if } \mathsf{up}_x(v) \text{ is } x :\bowtie c \\ \{\alpha \mid \alpha \bowtie v(y)+c\} & \text{if } \mathsf{up}_x(v) \text{ is } x :\bowtie y+c \end{cases}$
For example, it is possible to decrement the value of a clock by 1, or to set a clock non-deterministically at a value less than 2.

This model is very general and it is easy to prove that the reachability problem is not decidable for the whole class of updatable timed automata, by reducing the computation of a two counter machine to the computation of an updatable timed automaton (decrementation (resp. incrementation) of counters are simulated by decrementation (resp. incrementation) of clocks). In [20], tighter undecidable classes and several decidable classes are described. We will not enter into details here, but will present two undecidability proofs and describe one decidable class.

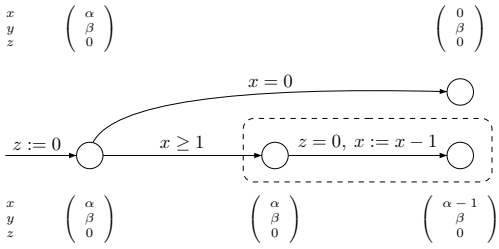
Decrementing clocks leads to undecidability. We now sketch the reduction from a two counter machine to updatable timed automata with resets to zero and decrementation. Let us consider a two counter machine $\mathcal{M}$ with the two counters c and d. We will construct a timed automaton $\mathcal{A}$ (with decrementations and resets to zero) such that the computation of $\mathcal{M}$ terminates if and only if a given state of $\mathcal{A}$ is reachable. The value of counter c (resp. counter d) is encoded by the value of clock x (resp. clock y). An additional clock z is used to rhythm the computation of automaton $\mathcal{A}$. Incrementation (and decrementation) of counters are simulated as follows.

- **Incrementation of counter c.**



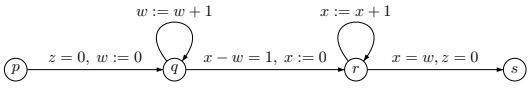
  For incrementing counter c, we let time elapse during one unit of time. The two clocks x and y thus increase by 1. It is then sufficient to decrease clock y by 1: the value of x in ℓ' is equal to the value of x in ℓ plus 1 whereas the value of y in ℓ' is equal to the value of y in ℓ. This correctly encodes an incrementation of c by 1.

- **Decrementation of counter c.**



  An explanation similar to the one for decrementation can be done.

Incrementing clocks also leads to undecidability as soon as diagonal constraints are used... From the previous reduction, it is sufficient to be able to simulate the part of the automaton which is framed with dashed lines, thus to decrease the value of a clock (say x) by 1.



It is easy to see that this module simulates an incrementation.

... but remains decidable when no diagonal constraints are used. We will see that the usual (diagonal-free) region partitioning is correct when also using incrementation of clocks. However this requires a more involved explanation. Indeed, the three conditions ①, ② and ③ are no more sufficient because more general operations on clocks are used. More precisely, we need to replace condition ③ by the following condition (where $\mathcal{R}$ is a finite partitioning of the set of valuations, and $\mathcal{U}$ is a finite set of updates):

③' We say that $\mathcal{R}$ is *compatible with updates in* $\mathcal{U}$ whenever for all $R, R' \in \mathcal{R}$, for each $\mathsf{up} \in \mathcal{U}$,if for some valuation $v \in R$, $\mathsf{up}(v) \cap R' \neq \emptyset$, then for every valuation $v' \in R$, $\mathsf{up}(v') \cap R' \neq \emptyset$.

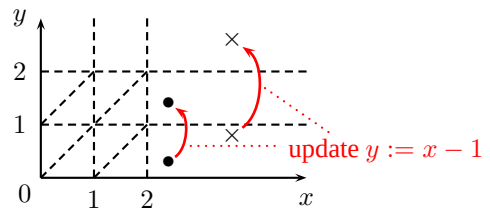
It is just an extension of Proposition 1 to prove that if, for a finite set of constraints $\mathcal{C}$ and a finite set of updates $\mathcal{U}$, we can construct a set of regions satisfying conditions ①, ② and ③', then the region automaton construction can be used to verify reachability (or more generally time-abstract invariant) properties.

Let us fix a finite set $\mathcal{C}$ of diagonal-free constraints, and a finite set of updates $\mathcal{U}$ of the form $x := y + c$ and possibly some resets of clocks. If the system of inequations

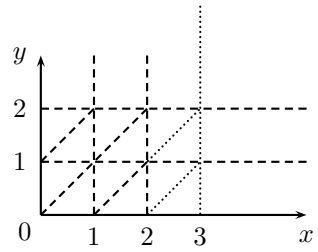
$$\{\alpha_x \geq c \mid (x \bowtie c) \text{ is in } \mathcal{C}\}$$
$$\cup \{\alpha_x \leq \alpha_y + c \mid (x := y + c) \text{ is in } \mathcal{U}\}$$

has a solution $(m_x)_{x \in X}$, then the diagonal-free set of regions where the maximal constant for x is m_x satisfies the three above-mentioned conditions. Note that if only updates of the form $x := x + 1$ are authorized then, as claimed before, the usual region partitioning is correct (because constraints $\alpha_x \leq \alpha_x + 1$ are trivially true).

However the usual region partitioning needs sometimes to be refined a little bit. Consider the following example: the maximal constant to which the two clocks x and y are compared is 2, both resets of x and y are allowed, and the more elaborated update $y := x - 1$. The system of inequations is $\{\alpha_x \geq 2, \alpha_y \geq 2, \alpha_y \leq \alpha_x - 1\}$. It has a solution, *eg* $\alpha_x = 2$ and $\alpha_y = 3$. We explain the intuition behind these conditions on **Fig.** 6.



(a) Classical partitioning not compatible with $y := x - 1$



(b) Set of regions satisfying conditions ①, ② and ③'.

Fig. 6. Partitioning for updates $y := x - 1$

Updatable timed automata have been studied in details in [20], where the precise frontier between decidable and undecidable subclasses has been depicted: among other results, when only diagonal-free constraints are used, decrementation of clocks leads to undecidability whereas incrementation leads to decidability, which may appear as a surprising result. It has also been proved that for every updatable timed automaton belonging to some decidable subclass, we can construct a timed automaton with silent actions (but with an exponential complexity blowup) which recognizes the same timed language.

5.5 Partial Conclusion

We have shortly presented in this section several extensions and variants of timed automata, having in mind the decidability of reachability checking. Many other extensions or subclasses could have been presented as well, for example timed automata with modulo constraints [23], or timed automata with event-predicting or event-recording timed automata [9, 35].

Historically, (linear) hybrid automata [30, 32] have not been defined and studied as an extension of timed automata, but they can be viewed as such. A hybrid automaton is roughly a timed automaton where variables (instead of clocks) grow in every

state following some differential equation. Linear hybrid automata are particular hybrid automata where variables evolve following linear differential equations. As soon as a variable has two different slopes, the hybrid automata model is undecidable [32]. In particular, *stopwatch automata, i.e.* timed automata in which clocks can be stopped, are undecidable. However, a decidable subclass has been exhibited, the so-called initialized rectangular automata. Hybrid automata are a very interesting model which would require a whole tutorial in itself. We better refer to [40] for an introduction to this model.

6 Algorithmics & Implementation

In practice the region automaton construction is not used in tools. Algorithms for "minimizing" the region automaton have been proposed for example in [3, 4, 43]. However in practice *on-the-fly* technics are preferred.

6.1 Reachability Analysis: Two Methods

There are two main families of (semi-)algorithms for analyzing reachability properties of systems (not only timed systems, but all kinds of systems).

Forward analysis. The general idea of forward analysis is to compute configurations which are reachable from initial configurations within 1 steps, 2 steps, etc... until final states are reached or until the computation terminates.

Backward analysis. The general idea of backward analysis is to compute configurations from which we can reach final configurations within 1 step, 2 steps, etc... until initial configurations are reached or until the computation terminates.

These two generic approaches are used for many models, for example counter machines, hybrid systems, etc... Of course, given a class of systems, specific technics (*e.g.* abstractions, widening operations, etc...) can be used for improving the computation. We will study how these approaches can be used for verifying timed automata.

6.2 Reachability Analysis in Timed Automata: Zones

We need now to look carefully at how the abovementioned general methods can be used for verifying timed automata. In particular, as timed automata have an infinite number of configurations, we need to use symbolic representations for doing the computation. Given a transition e of a timed automaton $\ell \xrightarrow{g,a,Y} \ell'$, we need to be able to compute, given a set W of valuations, both sets

$$\{v' \mid \exists v \in W\ \exists t \in \mathbb{T} \text{ s.t. } v' = [Y \leftarrow 0](v+t)\}$$

$$\{v \mid \exists v' \in W\ \exists t \in \mathbb{T} \text{ s.t. } [Y \leftarrow 0](v+t) = v'\}$$

It is worth to notice that if the forward computation starts in an initial state with all clocks initialized to 0 or if the backward computation starts from the final states with clocks set to any value (which is sufficient as we are only interested in reachability of discrete states), sets of valuations which are computed are *zones, i.e.* sets of valuations defined by a general clock constraint. Recall that general clock constraints are defined by the grammar:

$$g \ ::= \ x \bowtie c \ \mid \ x - y \bowtie c \ \mid \ g \wedge g$$

where $c \in \mathbb{Z}$, $\bowtie \in \{\leq, <, =, >, \geq\}$ and $x,\ y$ are clocks. A clock constraint g defines a zone $[\![g]\!] = \{v \in \mathbb{T}^X \mid v \models \varphi\}$. For analyzing timed automata, zones are the *symbolic representation* which is commonly used. For implementing forward and backward analysis, we need to be able to perform several operations on zones. From what has been said before, these operations are the following (Z and Z' are supposed to be zones):

- *Future of Z:* $\overrightarrow{Z} = \{v+t \mid v \in Z \text{ and } t \in \mathbb{T}\}$
- *Past of Z:* $\overleftarrow{Z} = \{v-t \mid v \in Z \text{ and } t \in \mathbb{T}\}$
- *Intersection of Z and Z':* $Z \cap Z' = \{v \mid v \in Z \text{ and } v \in Z'\}$
- *Reset to zero of Z w.r.t. set of clocks Y:* $[Y \leftarrow 0]Z = \{[Y \leftarrow 0]v \mid v \in Z\}$
- *Inverse reset to zero of Z w.r.t. set of clocks Y:* $[Y \leftarrow 0]^{-1}Z = \{v \mid [Y \leftarrow 0]v \in Z\}$
- *Test emptiness of Z:* decide whether $Z = \emptyset$

Using these operations, the basic steps of the forward and the backward computations can be rewritten as:

$$\begin{cases} \mathsf{Post}_e(Z) = [Y \leftarrow 0](\overrightarrow{Z} \cap [\![g]\!]) \\ \mathsf{Pre}_e(Z) = \overleftarrow{[Y \leftarrow 0]^{-1}(Z \cap [\![Y = 0]\!]) \cap [\![g]\!]} \end{cases}$$

6.3 The DBM Data Structure

For representing zones, the most common data structure which is used is the so-called DBM data structure (where DBM stands for "Difference Bounded Matrice"). This data structure has been

first introduced in [17] and then proposed in the framework of timed automata in [28]. Several presentations of this data structure can be found in the literature, for example in [24, 14, 18].

A *difference bounded matrice* (say *DBM* for short) for a set $X = \{x_1, \ldots, x_n\}$ of n clocks is an $(n+1)$-square matrice of pairs

$$(m; \prec) \in \mathbb{V} = (\mathbb{Z} \times \{<, \leq\}) \cup \{(\infty; <)\}.$$

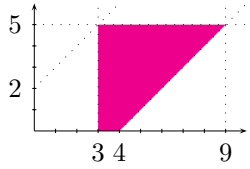
A DBM $M = (m_{i,j}, \prec_{i,j})_{i,j=1\ldots n}$ defines the following subset of $\mathbb{T}^n$ (the clock x_0 is supposed to be always equal to zero, *i.e.* for each valuation v, $v(x_0) = 0$):

$$\{v : X \to \mathbb{T} \mid \forall\, i, j,\ v(x_i) - v(x_j) \prec_{i,j} m_{i,j}\}$$

where $\gamma < \infty$ simply means that γ is some real without bound. This subset of $\mathbb{T}^n$ is a zone and will be denoted, in what follows, by $[\![M]\!]$. In what follows, to simplify notations, we will assume that all constraints are non-strict, so that coefficient of DBMs will be elements of $\mathbb{Z} \cup \{\infty\}$.

Example 3 *Consider the zone defined by the constraints* $(x_1 \geq 3) \wedge (x_2 \leq 5) \wedge (x_1 - x_2 \leq 4)$. *This zone, depicted below on the right, can be represented by the DBM below (on the left).*

$$\begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \end{array} \begin{array}{c} \begin{array}{ccc} x_0 & x_1 & x_2 \end{array} \\ \begin{pmatrix} \infty & -3 & \infty \\ \infty & \infty & 4 \\ 5 & \infty & \infty \end{pmatrix} \end{array}$$



A zone can have several representations using DBMs. For example, the zone of the previous example can equivalently be represented by the DBM

$$\begin{pmatrix} 0 & -3 & 0 \\ 9 & 0 & 4 \\ 5 & 2 & 0 \end{pmatrix}$$

A normal form can be defined on DBMs, which tightens all possible constraints. This can be done using a Floyd algorithm on the matrice (viewed as a weighted graph). A zone has a unique representation as a DBM in normal form. Tests like emptiness checking, or comparison of zones can then be done syntactically on the DBMs in normal form. For example, a zone Z is included in a zone Z' if the DBM in normal form representing Z is smaller than the DBM in normal form representing Z'. Finally all operations on zones described in section 6.2 can easily be done on the DBMs, details can be found in all mentioned papers on DBMs.

Let us just mention that the DBM data structure is the most basic data structure which is used for analyzing timed systems, some more involved BDD-like data structures can also be used, for example CDDs (which stands for "Clock Difference Diagrams") [37].

6.4 Backward Analysis

Let $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ be a timed automaton. Backward analysis then consists in computing the following sets of symbolic configurations: $\mathcal{S}_0 = \{(f, \mathbb{T}^X) \mid f \in F\}$, and iteratively $\mathcal{S}_{p+1} = \{(\ell, Z) \mid \exists e = (\ell \xrightarrow{g,a,Y} \ell')\exists (\ell', Z') \in \mathcal{S}_p \text{ s.t. } Z = \mathsf{Pre}_e(Z')\}, \ldots$

Theorem 4 *The backward computation terminates and is correct w.r.t. reachability, i.e. if a state is found reachable by the computation, then it is really reachable.*

Correctness is immediate as the computation is *exact* (as opposed to over-(or under-)approximate). Termination needs some additional argument, related to properties of the region partitioning associated with timed automata. The termination proof then relies on the following lemma, which can be proved as an exercise.

Lemma 2 *Let $\mathcal{A}$ be a timed automaton and let $\mathcal{R}$ be a set of regions satisfying conditions ①, ② and ③ (for $\mathcal{A}$). Consider a finite union of regions $\bigcup_{i=1}^{p} R_i$ (with $R_i \in \mathcal{R}$ for $1 \leq i \leq p$). Then the following holds:*

- *$\overleftarrow{\bigcup_{i=1}^{p} R_i}$ is a finite union of regions*
- *$[Y \leftarrow 0]^{-1}(\bigcup_{i=1}^{p} R_i)$ is a finite union of regions (for any set of clocks Y)*
- *$g \cap (\bigcup_{i=1}^{p} R_i)$ is a finite union of regions if g is a constraint of $\mathcal{A}$ (thus compatible with $\mathcal{R}$)*

Backward analysis thus appears as a very interesting method for analyzing timed systems. However, in practice, most commonly used tools (for example UPPAAL) prefer using a forward analysis procedure. A natural question then arises: what's the problem with backward analysis? It comes from the fact that the use of bounded integer variables really improves and eases the modeling of real systems. Backward analysis is then not suitable for arithmetical operations: for example if we know

in which interval lies the variable i and if we know that i is assigned the value $j.k + \ell.m$, it is not easy to compute the possible values of variables j, k, ℓ, m (apart from listing all possible tuples of values). For this kind of operations, forward analysis is much more suitable.

6.5 Forward Analysis

Let $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ be a timed automaton. Forward analysis then consists in computing the following sets of symbolic configurations: $\mathcal{S}_0 = \{(i, \mathbf{0}) \mid i \in I\}$, and then iteratively $\mathcal{S}_{p+1} = \{(\ell', Z') \mid \exists e = (\ell \xrightarrow{g,a,Y} \ell') \exists (\ell, Z) \in \mathcal{S}_p \text{ s.t. } Z' = \mathsf{Post}_e(Z)\}$, ...The forward analysis gives a correct answer (if it gives an answer), but may not terminate. An example of automaton where the forward computation does not terminate is given on **Fig.** 7. The zones which are computed are represented on the right part of the figure, and it is easy to check that the computation will never terminate.

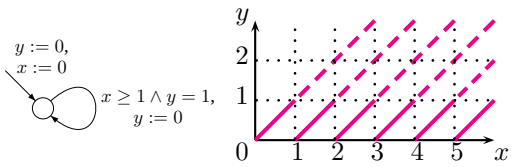


Fig. 7. Forward computation does not always terminate...

To overcome this problem, it is necessary to use some abstractions, several are proposed in [26]. For example, if Z and Z' are computed for the location ℓ, zones are replaced by the smallest zone containing both Z and Z': this approximation is called the "*convex-hull*"[3], it does not ensure termination and is only semi-correct w.r.t. reachability in the sense that a state which is announced as reachable may not be reachable. The most interesting abstraction studied in this paper is the *extrapolation* operator.

The extrapolation operator. The abstraction operator which is commonly used is called *extrapolation*, and sometimes *normalization* [14]. We will note it here Approx_k, it is defined up to a constant k as follows: if Z is a zone, $\mathsf{Approx}_k(Z)$ is the smallest k-bounded zone[4] which contains Z.

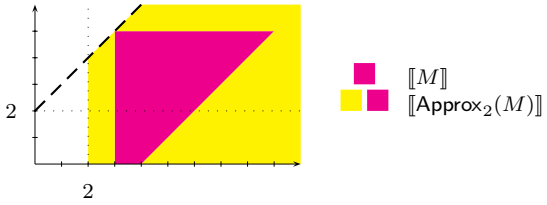
[3] It is a language abuse, because it is not reaaly the convex hull of the two zones, but it is the smallest zone containing the convex-hull of the two zones.

[4] A k-bounded zone is a zone defined by a k-bounded clock constraint.

This operation is well-defined on DBMs: if M is a DBM in normal form representing Z, a DBM representing $\mathsf{Approx}_k(Z)$ is M' where each coefficient less than $-k$ is replaced by $-k$ and all coefficients greater than k is replaced by $+\infty$, all other coefficients remain unchanged.

Example 4 *Consider the zone M of Example 3. Its extrapolation w.r.t. 2 is the following DBM:*

$$\mathsf{Approx}_2(M) = \begin{pmatrix} 0 & -2 & 0 \\ 9 & 0 & +\infty \\ +\infty & 2 & 0 \end{pmatrix}$$



Obviously,

- Approx_k is a finite abstraction operator because there are finitely many DBMs whose coefficients are either $+\infty$ or some integer between $-k$ and $+k$
- the computation of Approx_k is effective and can be done easily on DBMs
- Approx_k is a complete abstraction w.r.t. reachability because for every zone Z, $Z \subseteq \mathsf{Approx}_k(Z)$

The only problem stands in the correctness of Approx_k w.r.t. reachability: we have to find a constant k such that this abstraction operator will be correct w.r.t. reachability.

Theorem 5 *Let $\mathcal{A}$ be a **diagonal-free** timed automaton. Take k the maximal constant appearing in the constraints of $\mathcal{A}$. Then Approx_k is correct w.r.t. reachability in $\mathcal{A}$.*

Two different proofs of this theorem can be found in [18] and [12]. Note that this theorem does not extend to timed automata with general clock constraints. See [18] for a counter-example, and [21] for a solution to the problem.

6.6 Tools for Timed Systems

Several tools implement timed (and hybrid) automata.

- HyTech [31] is a model-checker for linear hybrid automata. Exact backward and

forward computations can be done, reachability properties can thus be checked (but there is of course no guarantee the computation will terminate). Many other operations on polyhedra can be performed, for example hiding of variables (corresponding to projections), "`while`" loops, emptiness checks, etc... HYTECH, which has been developed in Berkeley (USA), can be downloaded on

`http://www-cad.eecs.berkeley.edu:80/~tah/HyTech/`

- KRONOS [25] is a model-checker for timed automata. Exact as well as abstract backward and forward computations can be done. A backward procedure for the logic TCTL [2] is also implemented [34]. The tool KRONOS, which has been developed in Grenoble (France), can be downloaded on

  `http://www-verimag.imag.fr/TEMPORISE/kronos/`

- UPPAAL [38] is a model-checker for timed automata which performs forward analysis with extrapolation. It can verify reachability properties of timed systems with some extra features as bounded integer variables and broadcast channels. The tool UPPAAL, which is jointly developed in Aalborg University (Denmark) and Uppsala University (Sweden), can be downloaded on

  `http://www.uppaal.com/`

7 Conclusion

In this tutorial we have presented the basic model of timed automata, introduced at the beginning of the 90's by Rajeev Alur and David Dill [7]. One of the most important and most fundamental construction which is used in this domain is the region automaton construction: it finitely abstracts behaviours of timed automata into behaviours of finite automata, which allows to model-check many properties: although we only presented how reachability properties could be checked, properties in TCTL can also be verified using a region-like construction [2]. We have also presented several extensions of timed automata, concentrating on the decidability of the model-checking of reachability properties.

There are so many works which have been devoted to timed systems in general, and timed automata in particular, that it is hopeless to present the whole theory of timed automata in a single tutorial. The current tutorial presents some results on timed automata, focusing on the decidability of reachability properties and on implementation issues for verifying such properties.

References

[1] L. Aceto and F. Laroussinie. Is your model-checker on time ? on the complexity of model-checking for timed modal logics. *Journal of Logic and Algebraic Programming*, 52–53:7–51, 2002.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[3] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. 13th IEEE Real-Time Systems Symp. (RTSS'92)*, pages 157–166. IEEE Comp. Soc. Press, 1992.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proc. 3rd Int. Conf. Concurrency Theory (CONCUR'92)*, volume 630 of *LNCS*, pages 340–354. Springer, 1992.

[5] R. Alur, C. Courcoubetis, and T. A. Henzinger. The observational power of clocks. In *Proc. 5th Int. Conf. Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 162–177. Springer, 1994.

[6] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. 17th Int. Coll. Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[7] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[8] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[9] R. Alur, L. Fix, and T. A. Henzinger. A determinizable class of timed automata. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 1–13. Springer, 1994.

[10] R. Alur and P. Madhusudan. Decision problems for timed automata. In *Proc. 4th Int. School Formal Methods for the Design of Computer, Communication and Software Systems: Real Time (SFM-04:RT)*, volume 3142 of *LNCS*, pages 122–133. Springer, 2004.

[11] E. Asarin. Challenges in timed languages: From applied theory to basic theory. *The Bulletin of the European Association for Theoretical Computer Science*, 83, 2004.

[12] G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *Proc. 9th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 254–277. Springer, 2003.

[13] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proc. 4th Int. Work. Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *LNCS*, pages 147–161. Springer, 2001.

[14] J. Bengtsson. *Clocks, DBMs ans States in Timed Systems*. PhD thesis, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2002.

[15] B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2–3):145–182, 1998.

[16] B. Bérard and C. Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1–2):1–7, 2000.

[17] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In *Proc. IFIP 9th World Computer Congress*, volume 83 of *Information Processing*, pages 41–46. North-Holland/ IFIP, 1983.

[18] P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[19] P. Bouyer and F. Chevalier. On conciseness of extensions of timed automata. *Journal of Automata, Languages and Combinatorics*, 2005. To appear.

[20] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2–3):291–345, 2004.

[21] P. Bouyer, F. Laroussinie, and P.-A. Reynier. Diagonal constraints in timed automata — Forward analysis of timed systems. In *Proc. 3rd Int. Work. Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, LNCS. Springer, 2005. To appear.

[22] T. Brihaye, V. Bruyère, and J.-F. Raskin. Model-checking for weighted timed automata. In *Proc. Joint Conf. Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System (FORMATS+FTRTFT'04)*, volume 3253 of *LNCS*, pages 277–292. Springer, 2004.

[23] C. Choffrut and M. Goldwurm. Timed automata with periodic clock constraints. *Journal of Automata, Languages and Combinatorics*, 5(4):371–404, 2000.

[24] E. Clarke, O. Grumberg, and D. Peled. *Model-Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[25] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proc. Hybrid Systems III: Verification and Control (1995)*, volume 1066 of *LNCS*, pages 208–219. Springer, 1996.

[26] C. Daws and S. Tripakis. Model-checking of real-time reachability properties using abstractions. In *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.

[27] F. Demichelis and W. Zielonka. Controlled timed automata. In *Proc. 9th Int. Conf. Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 455–469. Springer, 1998.

[28] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. of the Work. Automatic Verification Methods for Finite State Systems (1989)*, volume 407 of *LNCS*, pages 197–212. Springer, 1990.

[29] T. A. Henzinger. Hybrid automata with finite bisimulations. In *Proc. 22nd Int. Coll. Automata, Languages and Programming (ICALP'95)*, volume 944 of *LNCS*, pages 324–335. Springer, 1995.

[30] T. A. Henzinger. The theory of hybrid automata. In *Proc. 11th Ann. Symp. Logic in Computer Science (LICS'96)*, pages 278–292. IEEE Comp. Soc. Press, 1996.

[31] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model-checker for hybrid systems. *Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[32] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[33] T. A. Henzinger, P. W. Kopke, and H. Wong-Toi. The expressive power of clocks. In *Proc. 22nd Int. Coll. Automata, Languages and Programming (ICALP'95)*, volume 944 of *LNCS*, pages 417–428. Springer, 1995.

[34] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[35] T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *Proc. 25th Int. Coll. Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 580–591. Springer, 1998.

[36] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[37] K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[38] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[39] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall Int., 1967.

[40] J.-F. Raskin. *An Introduction to Hybrid Automata*, chapter Handbook of Networked and Embedded Control Systems, pages 491–518. Springer, 2005.

[41] A. Robin. Aux frontières de la décidabilité... Master's thesis, DEA Algorithmique, Paris, 2004.

[42] S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Proc. 1st Int. Work. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 182–188. Springer, 2003.

[43] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

Vérification de programmes synchrones avec Lustre/Lesar

Pascal Raymond
Verimag-CNRS
2 av de Vignate
38610 Gières
Pascal.Raymond@imag.fr

Résumé

L'approche synchrone [3, 10] a été proposée dans les années 80 dans le but de concilier la programmation concurrente et le déterminisme. Le langage Lustre découle de cette approche. Il propose un style de programmation flot-de-données, dans la lignée de modèles classiques (block diagrams, circuits séquentiels). La sémantique formelle du langage a permis d'appliquer des méthodes de vérification formelle de la fonctionnalité. Dans les années 90, des méthodes de preuve par exploration de modèle (model-checking) ont été utilisées avec succès dans des domaine comme la vérification de protocoles ou celle des circuits logiques. Ces méthodes ont été adaptées pour la validation des programmes Lustre, et un model-checker spécifique (Lesar) a été développé.

1. Approche synchrone

1.1. Le domaine des systèmes réactifs

Le domaine d'application est celui des systèmes réactifs critiques (contrôle/commande, embarqué).

Le fonctionnement abstrait d'un système réactif est schématisé par la figure 1 :

- il réagit aux entrées fournies par son environnement en produisant des sorties,
- il est lui-même conçu de manière hiérarchique, comme un ensemble de sous-systèmes réactifs fonctionnant en parallèle.

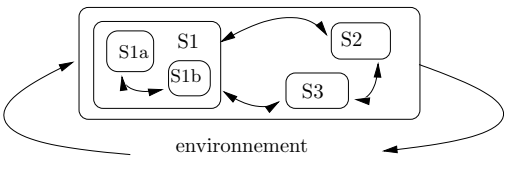


Figure 1. Un système réactif général

1.2. L'approche synchrone

L'approche classique pour l'implémentation d'un tel système est de transformer le parallélisme de description en parallélisme d'exécution, ce qui implique :

- la nécessité d'un d'exécutif complexe (OS temps-réel, primitive de communication/synchronisation),
- la difficulté de garantir a priori une sémantique globale (problème d'indéterminisme).

L'approche synchrone pour la programmation et l'implémentation de tels systèmes a été proposée pour concilier la conception hiérarchique et parallèle avec le déterminisme.

Hypothèse de travail : Le programmeur conçoit le système *comme si* les communications et les réactions se faisaient en *temps nul*. Il se concentre donc sur la *fonctionnalité* du système.

Validité de l'implémentation : Concrètement, la méthode garantit que les réactions se feront toujours en un *temps borné*, évaluable pour une architecture donnée.

1.3. Les langages synchrones

Plusieurs langages basés sur ce principe ont été proposés :

Lustre est un langage à flot de donnée, il a été transféré dans l'industrie via l'atelier SCADE[1].

Signal [12] est aussi un langage à flots de données, mais qui propose une notion d'horloge sophistiquée.

Esterel [4] est un langage à structures de contrôle "classiques". Il en existe une version graphique basées sur des automates hiérarchiques communiquants (SynchCharts [2]).

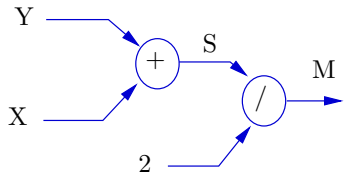
[1] `http://www.esterel-technologies.com/products/overview.html`

2. Le langage Lustre

2.1. Principes

Le langage Lustre [11] s'inspire du modèle classique des graphes à flots de données, où les applications sont vues comme des réseaux d'opérateurs connectés par des "fils" sur lesquels circulent des données (figure 2).

En première approximation, Lustre est donc un formalisme textuel permettant de décrire de tels graphes.




```
node Moyenne(X,Y:int)returns(M:int);
var S:int;
let
   M = S/2;
   S = (X+Y);
tel
```

Figure 2. Un graphe flot de données et son équivalent Lustre

Outre l'aspect flot de données, les caractéristiques du langage sont les suivantes :

Langage déclaratif : le corps d'un programme est un ensemble *d'équations*, dont l'ordre n'est pas significatif. Cela signifie que le langage respecte le principe de *substitution* : si `id = expr`, alors toute occurrence de `id` peut être remplacée par l'expression `expr`. Par exemple, dans la figure 2, on peut se passer de la variable intermédiare `S` et écrire simplement `M = (X+Y)/2`.

Sémantique synchrone : l'interprétation de la "circulation" des données se fait sur une *horloge discrète implicite*, qu'on assimile à l'ensemble des entiers naturels. Par exemple, `M` dénote une séquence infinie de valeurs entières $\mathtt{M}_0, \mathtt{M}_1, \mathtt{M}_2, \cdots$. La "traversée" des opérateurs se fait (sauf exception) en temps nul, d'où le terme de synchrone. Par exemple, `M = (X+Y)/2` signifie que $\forall t \in \mathbb{N}$, $\mathtt{M}_t = (\mathtt{X}_t + \mathtt{Y}_t)/2$.

Les opérateurs temporels : Pour pouvoir décrire des comportements temporel complexes, le langage propose un opérateur `pre` (pour previous) qui "décale" les flots dans le temps : $\forall t \geq 1 \quad (\mathtt{pre\ X})_t = \mathtt{X}_{t-1}$, et $(\mathtt{pre\ X})_0 = \perp$. L'opérateur `pre` est complété par un opérateur d'initialisation qui permet d'éviter les valeurs indéfinies : $(\mathtt{X\ ->\ Y})_0 = \mathtt{X}_0$ et $\forall t \geq 1 \quad (\mathtt{X\ ->\ Y})_t = \mathtt{Y}_t$. Par exemple, l'équation `N = 0 -> pre N + 1` définit la séquence des entiers naturels $(0, 1, 2, 3, \cdots)$.

Langage dédié : Le langage est dédié à la programmation des noyaux réactifs, et pas à la programmation des traitement complexes de données. Les types de base prédéfinis sont les booléens (`bool`), les entiers (`int`) et les flottants (`real`). Tous les opérateurs arithmétiques et logiques classiques sont prédéfinis. Les types plus complexes, ainsi que leurs opérateurs, sont vus en Lustre comme des types et des fonctions abstraits, et doivent être implémentés dans un langage hôte (typiquement C) [2].

Langage modulaire : Une fois défini par l'utilisateur, un programme Lustre (`node`) peut être réutilisé pour la construction d'un autre programme, au même titre qu'un opérateur prédéfini.

Remarque : le langage Lustre restreint aux booléens est équivalent au formalisme bien connu des *circuits séquentiels* : les opérateurs classiques (`and`, `or`, `not`) correspondent aux portes logiques, et la construction `false -> pre` correspond à la notion de registre.

2.2. Exemple du compteur de balises

Un train circule sur une voie où sont placées, de proche en proche, des *balises*. À l'intérieur du train, un *compteur de balise* reçoit un signal chaque fois qu'une `balise` est rencontrée, ainsi qu'un signal `seconde` provenant d'une horloge temps-réel.

La vitesse idéale étant de 1 balise par seconde, le compteur doit décider si le train est en avance, à l'heure, ou en retard. Pour éviter les oscillations trop rapides, on utilise un mécanisme d'hytérésis (décalage des "fronts" de changement d'état).

Le programme Lustre correspondant est présenté sur la figure 3.

3. Vérification de programme

Les méthodes qui vont être présentées ne s'appliquent bien sûr pas qu'à Lustre, mais plus généralement à tout langage ou formalisme reposant sur une notion de temps discret. C'est en particulier le cas pour les autres langages de la "famille" synchrone (Esterel, Signal etc.).

3.1. Notion de propriété temporelle

On s'intéresse ici à la vérification fonctionnelle : le programme calcule-t-il les bonnes sorties ? En

[2] Nous ne présentons ici que le langage "noyau" ; les versions actuelles, en particulier celle qui est utilisée dans l'atelier industriel Scade sont en fait beaucoup plus riches.

```
node compteur(sec,bal:  bool) returns (alheure,retard,avance:  bool);
var diff :  int;
let
   diff = (0 -> pre diff) + (if bal then 1 else 0) +
         (if sec then -1 else 0);
   avance = (true -> pre alheure) and (diff > 3)
         or (false -> pre avance) and (diff > 1);
   retard = (true -> pre alheure) and (diff < -3)
         or (false -> pre retard) and (diff < -1);
   alheure = not (avance or retard);
tel
```

Figure 3. Le programme compteur de balises

d'autres termes, le programme satisfait-il les propriétés qu'on attend de lui ? Ici, les propriétés attendues sont des relations temporelles entre les (séquences) d'entrées et de sorties ; on parle de *propriétés temporelles*.

3.2. Sûreté et vivacité

Il existe toute une théorie sur la classification des propriétés (et des logiques) temporelles. On retiendra pour ce cours uniquement une définition "intuitive" de la partition classique entre :

- les propriétés de sûreté (safety), qui expriment que quelque chose (de mauvais) n'arrive jamais,
- les propriétés de vivacité (liveness), qui exprime que quelque chose (de bon) peut ou doit arriver.

3.3. Exemple du compteur de balises.

Voici quelques exemples de propriétés qu'on peut attendre pour ce programme.

- On ne peut pas être en avance et en retard.
- On ne peut pas passer directement d'en retard à en avance (et réciproquement).
- On ne peut pas rester en retard pendant un seul instant.
- Si le train stoppe, il finira par être en retard.

Les trois premières propriétés sont des *safety*, tandis que la dernière est clairement une *liveness*.

3.4. Machine à mémoire

Le fonctionnement abstrait d'un programme Lustre (plus généralement d'un programme synchrone) est celui d'une machine à mémoire (figure 4) déterministe :

- l'état initial de la mémoire M est parfaitement déterminé,
- les valeurs des sorties et de l'état suivant de la mémoire sont fonctionnellement définies (par f et g) à partir des valeurs d'entrées et de l'état courant de la mémoire.

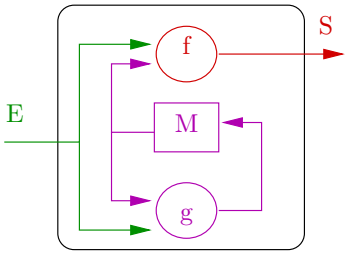


Figure 4. Machine à mémoire

3.5. Automate explicite

Une telle machine est aussi un *automate implicite*, en ce sens qu'elle est équivalente à un système à états/transitions (un automate), où chaque état correspond à une configuration particulière de la mémoire. La figure 5 représente une partie de cet automate ; notez que les transitions sont conditionnées par les valeurs des entrées, non représentées pour des raisons de lisibilité :

- toutes les transitions allant de gauche à droite sont conditionnées par `bal and not sec`,
- toutes les transitions allant de droite à gauche sont conditionnées par `not bal and sec`,
- dans tout état il existe une boucle (non représentée) conditionnée par `bal = sec`.

3.6. Principe du model-checking

L'automate explicite est un *modèle* qui synthétise *exactement* tous les comportements possibles du programme. Donc, explorer ce modèle permet d'étudier les propriétés du programme.

Le problème est que cet automate est en général infini, ou tout du moins énorme, ce qui rend impossible une exploration exhaustive.

L'idée est donc de travailler sur une abstraction finie (et pas trop grosse) de cet automate. Cette abstraction doit bien entendu conserver certaines propriétés du système initial (sinon elle serait totalement inutile).

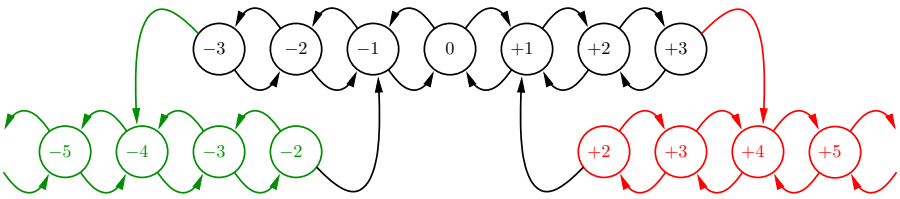


Figure 5. Les états explicites du compteur de balises

3.7. Exemple d'abstraction

L'abstraction booléenne consiste à abstraire tous ce qui n'est pas booléen (i.e. les mémoires et les calculs numériques). Dans l'exemple du compteur, cela revient à ignorer le compteur interne `diff`, et à introduire des "entrées" logiques à la frontière entre le monde booléen et le monde numérique (cf. figure 6).

- a_1 correspond à `diff > 3`,
- a_2 correspond à `diff < -3`,
- a_3 correspond à `diff < -1`,
- a_4 correspond à `diff > 1`,

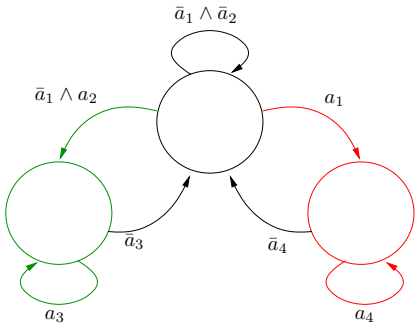


Figure 6. Une abstraction booléenne du compteur de balises

Cette abstraction ne représente plus exactement les comportements du programme, mais une sur-approximation de ceux-ci : certains comportements sont possibles sur l'approximation qui ne l'étaient pas sur le programme concret.

Cependant, certaines propriétés sont tout de même conservées :

- On ne peut pas être en avance et en retard (sûreté).
- On ne peut pas passer directement d'en retard à en avance (sûreté).

D'autres sont perdues :

- Si le train stoppe, il finira par être en retard (vivacité).
- On ne peut pas rester en retard pendant un seul instant (sûreté).

Plus grave, la négation de la dernière propriété, fausse sur le programme concret, a été introduite dans l'abstraction :

- On peut rester en retard pendant un seul instant (vivacité).

Il est donc très important de savoir quelles conclusions on a le droit de tirer de l'étude de l'abstraction.

3.8. Abstraction conservative et propriétés de sûreté

L'abstraction booléenne est un cas particulier de sur-approximation. Toute sur-approximation est *conservative* vis-à-vis de la classe des propriétés de sûreté : les propriétés de sûreté sont conservées ou perdues lors de l'abstraction, mais *jamais introduites*.

En conséquence, quand on vérifie une propriété de sûreté sur l'abstraction :

- la vérification aboutit, et la propriété est donc satisfaite,
- le vérification échoue et on ne peut (en général) pas conclure.

4. Expression des propriétés

4.1. Le schéma général du model-checking

Le model-checking est un domaine de recherche et d'application à part entière, sur lequel existe une abondante littérature. Les domaines d'application visés sont (historiquement) les protocoles de communications et les circuits logiques. Dans le cas de systèmes indéterministes, potentiellement bloquants, raisonner en terme d'ensemble de comportements ne suffit pas : il faut raisonner en terme *d'arbre d'exécution*. Ce domaine (lié à la notion de logique temporelle arborescente) ne sera pas abordé ici. On

se contentera du domaine de la *logique temporelle linéaire*.

Le principe "classique" est présenté sur la figure 7.

- On suppose disposer d'un modèle du système à vérifier ; ce modèle peut provenir d'une spécification de haut niveau (c'est souvent le cas dans le domaine des protocoles de communication), mais il peut aussi être obtenu automatiquement à partir d'une implémentation concrète (c'est en général le cas dans le domaine des circuits).
- On fournit la propriété attendue décrite dans une logique temporelle permettant de décrire des liveness et des safety (typiquement LTL).
- La propriété temporelle est (automatiquement) traduite dans une forme opérationnelle (typiquement un automate accepteur de la propriété). Notez que dans le cas des liveness, cette traduction nécessite des automates infinitaires (automate de Büchi).
- Le model-checking proprement dit consiste à explorer le produit des deux automates.

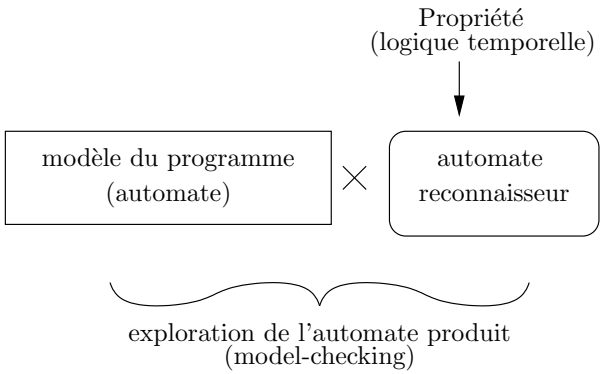


Figure 7. Schéma classique du model-checking

4.2. Model-checking des programmes synchrones

Le principe classique du model-checking a été adapté (et simplifié) au cas des programmes synchrones.

En premier lieu, la prise en compte des propriétés de vivacité est écartée, pour des raisons à la fois théoriques, pragmatiques et pratiques :

- comme on l'a vu précédemment, la vérification des programmes nécessite de faire des abstractions qui ne conservent pas les liveness,
- le domaine visé est celui des systèmes critiques où les safety sont primordiales, contrairement au liveness qui peuvent être jugées trop "vagues". Par exemple, la liveness :

  "*le train s'arrête inévitablement en cas de problème*"

  peut être renforcé en une safety plus précise :

  "*le train s'arrête inévitablement* **dans les 20 secondes** *en cas de problème*"

- les liveness nécessitent des techniques complexes, notamment l'apprentissage de logiques temporelles parfois difficiles à maîtriser.

4.3. Observateurs

L'abandon des propriétés de vivacité permet de s'affranchir des logiques temporelles au profit d'une méthode de description plus opérationnelle : celles des observateurs. Intuitivement, un observateur est un programme synchrone qui scrute les variables du programme et produit une sortie booléenne qui reste vraie *aussi longtemps* que la propriété attendue est satisfaite.

L'invariance (à vrai) de la sortie de cet observateur garantit que la propriété de sûreté sous-jacente est satisfaite.

L'avantage principal est que les propriétés peuvent être littéralement *programmées* dans le même langage que le système à valider. Cet argument pragmatique s'est avéré primordial pour la diffusion des méthodes formelles dans l'industrie, où l'apprentissage de nouveaux formalismes est généralement considéré comme rédhibitoire.

4.4. Exemples

Voici quelques exemples de traduction de propriétés en Lustre :

- On ne peut pas être en avance et en retard :
  `ok = not (avance and retard);`
- On ne peut pas passer directement d'en retard à en avance :
  `ok = true -> not ((pre retard) and avance);`
- On ne peut pas rester en retard pendant un seul instant :
  `ok = ((not PPretard) and Pretard) => retard;` avec :
  `Pretard = false -> pre retard;`
  `PPretard = false -> pre Pretard;`

4.5. Hypothèses

En général, un programme n'est censé fonctionner correctement que dans un environnement qui respecte un certain nombre de propriétés connues. De plus, il est souvent plus clair de décomposer une spécification selon le schéma classique hypothèse/conclusion.

Il est donc souhaitable (nécessaire) de pouvoir exprimer des *hypothèses* sous lesquelles la propriété doit être satisfaite.

Considérons par exemple la spécification suivante :

Si le train garde la vitesse idéale, il reste à l'heure

Dans ce cas la propriété est simplement `ok = alheure;`

L'hypothèse "garde la vitesse idéale" est plus compliquée à exprimer ; on peut se contenter d'une version très naïve (secondes et balises sont parfaitement synchrones) :

```
hypothese = (sec = bal);
```

Une interprétation plus réaliste consiste à dire que les secondes et les balises alternent :

```
hypothese = alterne(sec, bal);
```

où `alterne` est un programme Lustre accessoire que nous ne détaillerons pas.

Notez qu'un des intérêts pratiques des observateurs est justement de pouvoir définir ainsi des librairies d'observateurs standard, eux-mêmes validés formellement.

4.6. Model-checking des programmes synchrones

La figure 8 représente le principe de la vérification par observateurs. Le principe des observateurs est utilisé dans l'outil Lesar (model-checker de programmes Lustre), mais on le retrouve aussi dans d'autres outils (model-checker pour le langage Esterel, outil de preuve intégré à Scade). L'utilisateur doit construire un *programme de vérification*, qui intègre :

- le programme à valider proprement dit,
- l'observateur de la propriété attendue,
- l'observateur des hypothèses (qu'on appelle *assertion* en Lustre).

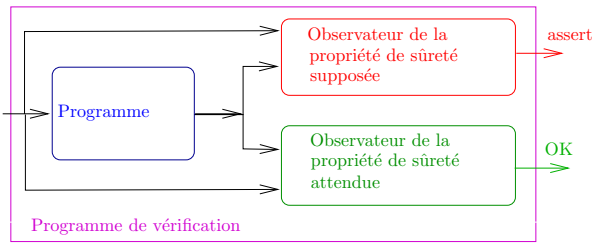


Figure 8. Programme de vérification

Le rôle du model-checker est alors :

- d'extraire une abstraction finie du programme de preuve (typiquement une abstraction booléenne),
- d'explorer cette abstraction finie en cherchant à établir que :
  quelque soit une séquence d'entrée (arbitrairement longue), aussi longtemps que la sortie `assert` reste vraie, alors la sortie `OK` reste vraie.

En terme de logique temporelle, la propriété complète est :

P1 : *(toujours* `assert`*)* $\Rightarrow$ *(toujours* `OK`*)*

Formellement, il ne s'agit pas d'une simple propriété de sûreté (i.e. un simple invariant), mais elle ne pose pas cependant pas de problème théorique : l'abstraction finie est conservative pour ce type propriété.

En fait, le problème est technique (algorithmique), car vérifier exactement ce type de propriété est plus coûteux que vérifier un simple invariant.

On se contentera de vérifier une *approximation* (conservative) de **P1** :

P2 : *toujours(*`assert` *n'a jamais été faux* $\Rightarrow$ `OK`*)*

5. Algorithmique

5.1. Automate booléen

On fait maintenant abstraction d'un langage de programmation particulier pour retenir un modèle simple, équationnel, des programmes de preuve.

Un programme de vérification est caractérisé par :

- un ensemble de variables libres (entrées) V et un ensemble de variables d'états (mémoires) S,
- l'état initial de la mémoire, caractérisé par une fonction $Init : \mathbb{B}^{|S|} \rightarrow \mathbb{B}$ (on peut considérer un ensemble d'états initiaux),
- des fonctions de transitions $g_k : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$, pour chaque $k = 1 \cdots |S|$,
- une fonction hypothèse $H : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$,
- et une fonction propriété $\Phi : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$.

Le passage d'un langage particulier comme Lustre vers ce modèle équationnel est trivial : les fonctions booléennes sont de simples expressions algébriques, extraites du programme.

5.2. Automate explicite associé

L'automate booléen "code" un système à états/transitions explicite :

- $Q = \mathbb{B}^{|S|}$ est l'espace d'états (l'ensemble des valeurs potentielles de la mémoire),
- $Init \subseteq Q$ est l'ensemble des états initiaux (on assimile fonctions caractéristiques et ensembles),

- la relation de transition $R \subseteq Q \times \mathbb{B}^{|V|} \times Q$ est définie par

$$(q, v, q') \in R \quad \Leftrightarrow \quad q'_k = g_k(q, v) \quad k = 1 \cdots |S|$$

on note $q \xrightarrow{v} q'$ pour $(q, v, q') \in R$.
Dans notre cas R est une *fonction* : q' est défini de manière unique pour tout couple (q, v). Par abus de langage, on appellera donc *transitions* les couples de $T = Q \times \mathbb{B}^{|V|}$.

- $H \subseteq T$ est l'ensemble des transitions qui satisfont l'hypothèse,
- $\Phi \subseteq T$ est l'ensemble des transitions qui satisfont la propriété.

5.3. Fonctions *pre* et *post*

Pour simplifier les données du problème, on va raisonner uniquement en terme d'état. Pour cela, on quantifie les variables libres et on intègre l'hypothèse dans les définitions suivantes ; pour tout état $q \in Q$, et tout ensemble d'états $X \subseteq Q$ on définit :

- $post_H(q) = \{q' \mid \exists v \; q \xrightarrow{v} q' \wedge H(q, v)\}$,
- $POST_H(X) = \bigcup_{q \in X} post_H(q)$
- $pre_H(q) = \{q' \mid \exists v \; q' \xrightarrow{v} q \wedge H(q', v)\}$,
- $PRE_H(X) = \bigcup_{q \in X} pre_H(q)$

5.4. Les états remarquables

A partir des états initiaux Init, on définit l'ensemble des états accessibles par le point fixe suivant :

$$\text{Acc} \quad = \quad \mu X \cdot (X = \text{Init} \cup POST_H(X))$$

On notera Acc_0 pour Init.

On définit l'ensemble des états d'erreur (immédiate) par :

$$\text{Err} \quad = \quad \{q \mid \exists v \; H(q, v) \wedge \neg\Phi(q, v)\}$$

c'est-à-dire l'ensemble des états dont au moins une transition sortante viole la propriété tous en satisfaisant l'hypothèse.

Par extension, on définit l'ensemble des "mauvais" états potentiels par le point fixe suivant :

$$\text{Bad} \quad = \quad \mu X \cdot (X = \text{Err} \cup PRE_H(X))$$

On notera Bad_0 pour Err.

5.5. Principe de l'exploration

Le but de la preuve est d'établir, d'une manière ou d'une autre, qu'aucun état accessible n'est mauvais ($\text{Acc} \cap \text{Bad} = \emptyset$).

Bien évidemment, il est inutile pour cela de calculer les deux points fixes. On peut se contenter d'établir indifféremment :

- que $\text{Acc} \cap \text{Bad}_0 = \emptyset$, c'est la méthode *en avant*,
- que $\text{Bad} \cap \text{Acc}_0 = \emptyset$, c'est la méthode *en arrière*.

Notez que dans notre cas (systèmes déterministes), il y a une dissymétrie entre les méthodes puisque la relation de transition est une fonction dans un sens (avant), mais est a priori quelconque dans l'autre (arrière).

6. Algorithmes énumératifs

6.1. En avant

L'algorithme en avant est un calcul de point fixe typique. On explore un par un les états accessibles, en vérifiant au fur et à mesure la validité de Φ. Si l'algorithme converge sans erreur, la propriété est établie.

```
CurAcc := Init
Done := empty
while it exits q in CurAcc - Done do {
      (* q ∈ CurAcc \ Done *)
      for all q' in post(q) do {
          if q' in Bad0 then EXIT(failed)
          put q' in CurAcc
      }
      put q in Done
}
(* we have CurAcc = Done = Acc *)
EXIT(succeed)
```

La figure 9 schématise, à gauche, le cas où la vérification réussit : l'exploration converge sans qu'un état de Err ne soit atteint. À droite, un état de Err est atteint en cours d'exploration : l'algorithme s'arrête sur un échec.

Notez qu'il ne s'agit ici que du principe général de l'algorithme. Dans une implémentation concrète il y beaucoup à faire pour rendre cet algorithmique un tant soit peu efficace (ou tout du moins pour limiter son inefficacité !). Quoiqu'il en soit, la complexité théorique est énorme : on doit explorer un nombre potentiellement exponentiel d'états ($2^{|S|}$), et pour chaque état un nombre potentiellement exponentiel de transitions ($2^{|V|}$).

Les méthodes énumératives sont donc limitées à des systèmes de taille relativement petites (quelques dizaines de variables en général).

6.2. En arrière

Il est tout à fait possible d'envisager une méthode énumérative en arrière. Cependant une telle méthode n'est jamais utilisée car encore plus inefficace que la méthode en avant. Le problème vient de la dissymétrie engendrée par le déterminisme.

L'algorithme en avant nécessite certes un nombre exponentiel de calculs, mais ces calculs sont relativement simples (application de fonctions booléennes).

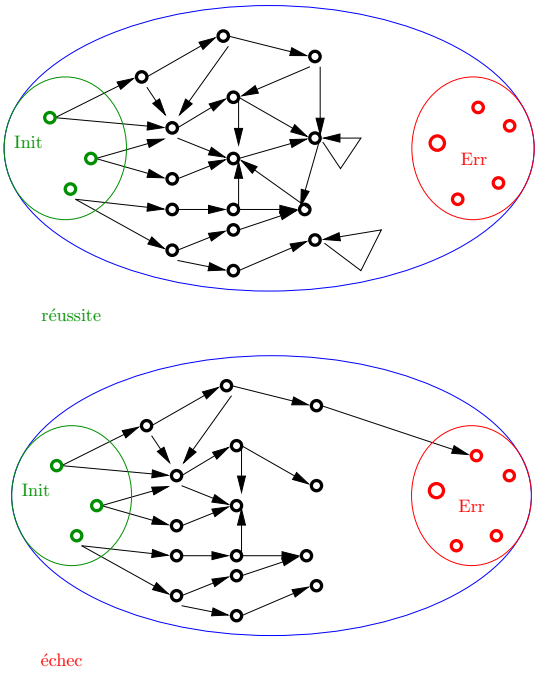


Figure 9. Exploration énumérative en avant

Avec une méthode en arrière, on a toujours un nombre exponentiel de calculs à effectuer, mais ces calculs sont très coûteux (résolution des quantificateurs existentiels).

7. Algorithmes symboliques

7.1. Principes

Les techniques de model-checking symbolique [7, 8, 9, 13] sont apparues dans les années 90, et sont dues à la convergence de deux facteurs essentiels :

- L'idée de s'affranchir de la taille concrète des ensembles en se basant sur des représentations *implicites* plutôt qu'explicites.
- La (re)découverte de techniques de représentation des prédicats logiques par des graphes de décision (BDD) qui ont permis de mettre en œuvre cette idée ([1, 6]).

Pour présenter et comprendre les algorithmes symboliques, il n'est pas nécessaire de savoir exactement comment fonctionnent les BDD. On retiendra simplement qu'ils permettent une représentation *canonique* et *souvent compacte* des prédicats booléens, et donc par extension, des ensembles de solutions de ces mêmes prédicats.

Par exemple, si x, y, z sont des variables d'états, le prédicat $(x \oplus y) \wedge \neg z$ "code" tous les états où z est faux, et où soit x, soit y est vrai.

Dans la suite, on manipulera des ensembles (d'états, de transitions) comme des objets atomiques. On supposera disponibles sur les ensembles/prédicats les opérations suivantes (dont on donne une idée de la complexité) :

- Tester l'égalité (et donc la différence) de deux ensembles ou prédicats est une opération "gratuite" (car built-in, cf. la canonicité). En particulier le problème de la décision est résolu par l'utilisation de BDD : un prédicat est satisfiable si et seulement si il n'est pas identiquement faux.
- La complémentation/négation est disponible et a un coût constant (négligeable).
- Les opérations binaires (union/disjonction, intersection/conjonction) sont disponibles et ont un coût polynomial en la taille des arguments (typiquement de l'ordre du produit des tailles des arguments).
- Par extension, les quantifications sont elles aussi disponibles (e.g. $\exists x\ f(x,y) = f(0,y) \vee f(1,y)$).

Notez que le passage d'une expression algébrique quelconque vers sa forme canonique sous forme de BDD a, dans le pire des cas, un coût exponentiel en temps et en mémoire.

7.2. En avant

Le schéma de l'algorithme est extrêmement simple : il manipule une variable A qui code l'ensemble des états accessibles connus. Cet ensemble est itérativement augmenté de ses successeurs directs. Si A intersecte les états d'erreur, la preuve échoue. Si le calcul converge sans intersecter Err, la preuve réussit.

En fait, la complexité réside essentiellement dans le calcul d'image (la fonction $POST_H$), dont on évoque l'implémentation plus loin :

A := Init
while true {
    if $A \cap$ Err $\neq \emptyset$ then EXIT(failed)
    $A' := A \cup POST_H(A)$
    if $A' == A$ then EXIT(succeed)
    else $A := A'$

La figure 10 représente, à gauche, le cas où la preuve réussit : après k itérations, on converge sans intersecter Err ; A contient alors l'ensemble Acc des états accessibles. À droite, la preuve échoue : Err est intersecté après k itérations.

Le schéma d'algorithme présenté est en fait relativement naïf : il procède strictement en largeur, et (re)calcule systématiquement l'image de tous les états connus.

Dans une implémentation concrète, l'algorithme d'exploration est en général beaucoup plus sophistiqué et cherche à trouver un compromis entre :

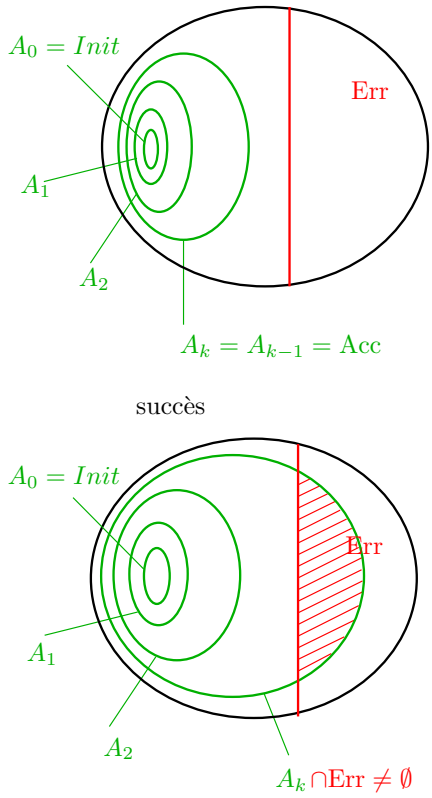


Figure 10. Exploration symbolique en avant

- le nombre d'itérations nécessaires (le nombre minimum d'itération est le diamètre du graphe d'état, et est obtenu par un parcours en largeur strict),
- la complexité des calculs d'image à chaque itération.

7.3. Calcul symbolique d'image

Le calcul symbolique de l'image ($POST_H$) consiste à construire la fonction caractéristique des états buts connaissant la fonction caractéristique des états sources (X).

Ce calcul est extrêmement coûteux, et des algorithmes relativement sophistiqués sont nécessaires pour atteindre une certaine efficacité. La présentation de tels algorithmes requiert une bonne connaissance de la représentation interne des formules (BDD).

On se contentera donc d'illustrer la faisabilité du calcul par un algorithme naïf (et inefficace). L'idée est simplement de montrer que le calcul d'image se ramène à une "simple" manipulation de formules logiques.

On va construire, dans un premier temps, un formule faisant intervenir :

- le vecteur des variables d'état source $s = (s_1, \cdots, s_n)$
- le vecteur des variables d'entrée $v = (v_1, \cdots, v_m)$
- le vecteur des variables d'état but $s' = (s'_1, \cdots, s'_n)$

Les étapes sont les suivantes :

- s est un état source :

$$X(s)$$

- et (s, v) satisfait l'hypothèse :

$$X(s) \wedge H(s, v)$$

- et chaque s'_i est l'image de (s, v) par la fonction de transition correspondante :

$$X(s) \wedge H(s, v) \wedge \bigwedge_{i=1}^{n} (s'_i = g_i(s, v))$$

À ce point, on a obtenu une "grosse" formule qui caractérise exactement les transitions $s \xrightarrow{v} s'$ telles que $s \in X$. Par quantification existentielle [3] des s et des v , on obtient une formule ne dépendant plus que des s' et qui caractérise exactement les états accessibles depuis X :

$$\exists s, v \;\; (X(s) \wedge H(s, v) \wedge \bigwedge_{i=1}^{n} (s'_i = g_i(s, v)))$$

8. En arrière

La forme générale est parfaitement duale à celle de l'algorithme en avant : il suffit d'échanger les rôles de Init et de Err, et de remplacer le calcul d'image par un calcul de prédécesseur (fonction PRE_H).

$B :=$ Err
while true {
    if $B \cap$ Init $\neq \emptyset$ then EXIT(failed)
    $B' := B \cup PRE_H(B)$
    if $B' == B$ then EXIT(succeed)
    else $A := A'$

Quand l'algorithme réussit, on a bien $B =$ Bad, c'est-à-dire l'ensemble de tous les états qui peuvent conduire à une erreur.

L'essentiel de la complexité réside dans le calcul d'image arrière (PRE). Comme dans le cas du $POST$, une implémentation simple, par manipulation d'opérateurs logiques sur les BDD, peut être implémentée. Elle est même relativement plus simple

[3] On rappelle que la quantification booléenne se ramène à des opérations algébriques simples selon le principe $\exists x \;\; f(x, y) = f(0, y) \vee f(1, y)$.

à comprendre puisqu'il s'agit de composer les fonctions de transitions par la fonction caractéristique des états buts.

Il est assez difficile de comparer dans l'absolu les algorithmes en avant et en arrière. On peut simplement dire, de manière empirique et statistique que les algorithmes en avant sont très souvent plus efficaces. Il existe cependant des cas particuliers où l'algorithme en arrière soit (bien) meilleur : c'est en particulier le cas où la propriété est *intrinsèquement* un invariant inductif, et qu'elle ne dépend donc pas d'un état initial particulier[4] .

9. Conclusion et travaux en relation

Bien que théoriquement décidable, la vérification des systèmes finis se heurte au problème de l'explosion combinatoire. Les méthodes symboliques, basées sur l'utilisation de BDD, permettent de s'affranchir de la taille concrête des modèles : la complexité n'est pas directement liée au nombre de variables, mais à la complexité des relations qui existent entre ces variables. Il n'en demeure pas moins que l'utilisation de BDD est extrêmement coûteuse en mémoire (exponentielle dans le pire des cas).

Les méthodes de décision de type SAT (pour *satisfiability*) consistent à énumérer les solutions d'une formule logique sans forcément les conserver en mémoire (ce que font les BDD). Elle ont un coût exponentiel en temps, mais polynomial en mémoire. Les algorithmes de model-checking peuvent être adaptés pour utiliser un moteur "SAT" à la place des BDD. Vérifier la convergence de l'exploration avec des méthode SAT s'avère en fait extrêmement coûteux, c'est pourquoi on se contente de vérifier la validité des propriétés pour des exécutions bornées dans le temps, on parle alors de *bounded model-checking*[5].

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[2] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sept. 1991.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In *Advances in Computers*, volume 58. Academic press, 2003.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, June 1990.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[9] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In R. Kurshan, editor, *International Workshop on Computer Aided Verification*, Rutgers (N.J.), June 1990.

[10] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[12] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, Sept. 1991.

[13] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer Aided Design (ICCAD)*, Nov. 1990.

[4] Une propriété des variables d'états P est un invariant inductif si $preP \Rightarrow P$.

Model Checking for Probabilistic Timed Automata with Digital Clocks*

Marta Kwiatkowska, Gethin Norman, David Parker
School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK
{mzk,gxn,dxp}@cs.bham.ac.uk

Jeremy Sproston
Dipartimento di Informatica, Università di Torino, 10149 Torino, Italy
sproston@di.unito.it

Abstract

Probabilistic timed automata, a variant of timed automata extended with discrete probability distributions, is a modelling formalism suitable for describing formally both nondeterministic and probabilistic aspects of real-time systems, and is amenable to model checking against probabilistic timed temporal logic properties. However, the previously developed verification algorithms either suffer from high complexity, give only approximate results, or are restricted to a limited class of properties. In the case of classical (non-probabilistic) timed automata it has been shown that for a large class of real-time verification problems correctness can be established using an integral model of time (digital clocks) as opposed to a dense model of time. Based on these results we address the question of under what conditions digital clocks are sufficient for the performance analysis of probabilistic timed automata and show that this reduction is possible for an important class of systems and properties including probabilistic reachability and expected reachability. We demonstrate the utility of this approach by applying the method to the performance analysis of the dynamic configuration protocol for IPv4 link-local addresses.

1 Introduction

Network protocols increasingly rely on the use of randomness and timing delays, for example the exponential back-off in Ethernet and IEEE 802.11. Since these protocols execute in a distributed environment, it is important to also consider nondeterminism when modelling their behaviour. A natural model for systems that exhibit nondeterminism, probability and real-time, called *probabilistic timed automata* – a probabilistic extension of timed automata [2] – has been proposed in [35]. In probabilistic timed automata, real-valued clocks measure the passage of time, and transitions can be probabilistic, that is, be expressed as a discrete probability distribution on the set of target states. In [35] model-checking algorithms for verifying the likelihood of certain temporal properties being satisfied by such system models are introduced. These model checking algorithms are either based on *region equivalence* [2], which results in prohibitively large state spaces for realistic systems, or on *forwards reachability*, which leads to approximate results [35, 18]. An alternative approach, based on *backwards reachability*, is given in [36, 37]; while this can be more efficient than the region equivalence approach and leads to exact results, the approach has been applied only to probabilistic temporal logics, and not to other classes of performance properties such as expected-time or expected-cost.

When modelling real-time systems there is often a trade off between expressiveness and complexity. For example, a *dense* time model is more expressive than an *integral* time model. However, an integral time model can be easier to verify, since it often leads to a finite-state system and allows one to apply the efficient symbolic methods developed for untimed systems. Henzinger et al. [28] study the question of when real-time properties can be verified using only integral durations (digital clocks), and show that such a reduction is possible for a large class of systems and properties, such as time-bounded invariance and response. Other related works include [6], where it was observed that to perform reachability analysis of certain classes of timed automata one needs only consider integer clock values, and [12, 13] which show that using BDDs and integral durations can lead to efficient methods for performing reachability analysis of timed automata. We also mention [23] and [39] which investigate the power of digital clocks.

The main contribution of this paper is to extend this direction of research to the domain of probabilistic timed automata by showing that digital clocks are sufficient for analysing a large class of probabilistic real-time systems and performance measures. The models that can be considered are those which can be represented by *closed*, *diagonal-free* probabilistic timed automata, intuitively automata whose clock constraints do not compare the values

*Supported in part by the EPSRC grants GR/N22960, GR/S11107 and GR/S46727, FORWARD and MIUR-FIRB Perf.

of clocks with one another or contain strict comparisons with constants. The performance measures include *probabilistic reachability* properties, which for example allow us to check the correctness of the following statements: 'with probability 0.05 or less, the system aborts' and 'with probability 0.99 or greater, a data packet will be delivered within 5 time units'. Additionally, *expected reachability* properties can be verified using digital clocks, which enable us to validate statements such as: 'the expected time until a data packet is delivered is at most 20ms', 'the expected number of packets sent before failure is at least 100' and 'the expected number of retransmissions before a packet is sent is at most 5'.

We then demonstrate the applicability of this approach on a case study using the probabilistic symbolic model checker PRISM [31, 41] to perform the analysis. In the protocol we study, the ZeroConf dynamic configuration protocol for IPv4 link-local addresses [17], the interplay between real-time, non-determinism and probabilistic behaviour is critical and it can be modelled naturally as a (closed, diagonal-free) probabilistic timed automaton. Preliminary results concerning this case study can be found in [33].

This paper is an abbreviated version of [34].

2 Preliminaries

2.1 Probability and Measure Theory

We assume some familiarity with probability and measure theory, see e.g. [24]. Consider a set Ω. A *σ-field* on Ω, denoted $\mathcal{F}$, is a family of subsets of Ω that contains Ω, and is closed under complementation and countable union. The elements of a σ-field are called *measurable sets*, and $(\Omega, \mathcal{F})$ is called a *measurable space*.

Definition 1 *Let $(\Omega, \mathcal{F})$ be a measurable space. A function $P : \mathcal{F} \to [0,1]$ is a* probability measure *on $(\Omega, \mathcal{F})$, and $(\Omega, \mathcal{F}, P)$ is a* probability space, *if P satisfies the following properties:*

1. *$P(\Omega) = 1$;*
2. *if $A_1, A_2, \ldots$ is a disjoint sequence of elements of $\mathcal{F}$, then $P(\cup_i A_i) = \sum_i P(A_i)$.*

The measure P is also referred to as a *probability distribution*. The set Ω is called the sample space, and the elements of $\mathcal{F}$ are called events.

Definition 2 *Let $(\Omega, \mathcal{F})$ and $(\Omega', \mathcal{F}')$ be two measurable spaces. A function $f : \Omega \to \Omega'$ is said to be a* measurable function *from $(\Omega, \mathcal{F})$ to $(\Omega', \mathcal{F}')$ if $f^{-1}(A') \in \mathcal{F}$ for all $A' \in \mathcal{F}'$.*

Theorem 3 ([15]) *Let $(\Omega, \mathcal{F})$ and $(\Omega', \mathcal{F}')$ be measurable spaces, and suppose that P is a measure on $(\Omega, \mathcal{F})$ and the function $T : \Omega \to \Omega'$ is measurable. If f is a real non-negative measurable function on $(\Omega', \mathcal{F}')$, then:*

$$\int_{\omega \in \Omega} f(T\omega)\, \mathrm{d}P = \int_{\omega' \in \Omega'} f(\omega')\, \mathrm{d}PT^{-1} \ .$$

A discrete probability *distribution* over a countable set Q is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$. For a possibly uncountable set Q', let $\mathsf{Dist}(Q')$ be the set of distributions over countable subsets of Q'. For $q \in Q$, let μ_q be the *point distribution* at q which assigns probability 1 to q.

2.2 Linear Programming

In this section we introduce some results of (integer) linear programming which will be required in Section 4.2.

Definition 4 *A matrix A is* totally unimodular *if each subdeterminant of A is 0, $+1$ or -1.*

Theorem 5 ([42] Theorem 19.3) *Let $\mathbf{A}$ be a matrix with entries 0, $+1$ and -1. Then the following are equivalent:*

1. *$\mathbf{A}$ is totally unimodular;*
2. *each collection of columns of $\mathbf{A}$ can be split into two parts so that the sum of the columns in one part minus the sum of the columns in the other part is a vector with entries only 0, $+1$ and -1.*

Theorem 6 ([42] Corollary 19.1.a) *Let $\mathbf{A}$ be a totally unimodular matrix, and let $\mathbf{b}$ and $\mathbf{c}$ be integral vectors. Then both problems in the linear programming duality equation*

$$\max\{\mathbf{cx} \,|\, \mathbf{x} \geq 0 \wedge \mathbf{Ax} \leq \mathbf{b}\} = \min\{\mathbf{yb} \,|\, \mathbf{y} \geq 0 \wedge \mathbf{yA} \geq \mathbf{c}\}$$

have integral optimum solutions.

2.3 Discrete-time Markov chains

We now introduce discrete-time Markov chains (DTMCs), a widely-studied stochastic process.

Definition 7 *A* DTMC *is a tuple $\mathcal{D} = (S, \overline{s}, \mathbf{P})$ where:*

- *S is a set of* states, *including the initial state $\overline{s}$;*
- *$\mathbf{P} : S \times S \to [0,1]$ is a* transition probability matrix, *such that for any $s \in S : \sum_{s' \in S} \mathbf{P}(s, s') = 1$.*

Each element $\mathbf{P}(s, s')$ of the transition probability matrix gives the probability of making a transition from state s to state s'. An execution of a system which is being modelled by a DTMC is represented by a *path*. Formally, a path ω is a non-empty finite or infinite sequence of states. In the case of a finite path $s_0 s_1 \cdots s_n$, we require $\mathbf{P}(s_i, s_{i+1})$ for all $0 \leq i < n$, whereas, for an infinite path $s_0 s_1 s_2 \cdots$, we require $\mathbf{P}(s_i, s_{i+1})$ for all $i \geq 0$. We denote by $\omega(i)$ the $(i+1)$th state of a path ω, $|\omega|$ the length of ω (number of transitions), and for a finite path ω, the last state by $last(\omega)$. Observe that a path can comprise of a single state, in which case its number of transitions is zero. We say that a finite path ω of length n is a *prefix* of the infinite path ω' if $\omega(i) = \omega'(i)$ for $0 \leq i \leq n$. Also, we use $\omega^{(k)}$ to denote the prefix of length k of ω. The sets of all finite and infinite paths starting in state s are denoted $Path_{fin}(s)$ and $Path_{ful}(s)$, respectively.

In order to reason about the probabilistic behaviour of the DTMC, we need to be able to determine the probability that certain paths are taken. This is achieved by defining, for each state $s \in S$, a probability measure $Prob_s$ over $Path_{ful}(s)$. Below, we give an outline of this construction. For further details, see [30]. The probability measure is induced by the transition probability matrix $\mathbf{P}$ as follows. First, for any finite path $\omega \in Path_{fin}(s)$ of length n, we define the probability $\mathbf{P}_s(\omega)$:

$$\mathbf{P}_s(\omega) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n = 0 \\ \mathbf{P}(\omega(0),\omega(1))..\mathbf{P}(\omega(n-1),\omega(n)) & \text{o/w.} \end{cases}$$

Next, we define the *cylinder set* $C(\omega)$ as:

$$C(\omega) \stackrel{\text{def}}{=} \{\omega' \in Path_{ful}(s) \mid \omega \text{ is a prefix of } \omega'\},$$

that is, the cylinder set $C(\omega)$ is the set of all infinite paths with prefix ω. Then let Σ_s be the smallest σ-algebra on $Path_{ful}(s)$ which contains all the sets $C(\omega)$, where ω ranges over paths in $Path_{fin}(s)$. We define the probability measure $Prob_s$ on Σ_s as the unique measure such that: $Prob_s(C(\omega)) = \mathbf{P}_s(\omega)$ for all $\omega \in Path_{fin}(s)$.

2.4 Timed probabilistic systems

We now introduce *timed probabilistic systems* which extend DTMCs by allowing both non-deterministic and probabilistic behaviour and in which transitions are labelled with either a duration taken from a time domain or an action. Timed probabilistic systems are an extension of Markov decision processes [22] and a variant of Segala's probabilistic timed automata [43].

Definition 8 *A* timed probabilistic system *is a tuple* $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$ *where:*

- S *is a set of* states*, including an* initial state $\bar{s} \in S$*;*
- $\mathcal{A}ct$ *is a finite set of* actions*;*
- $\mathbb{T} \subseteq \mathbb{R}$ *is a set of* durations*, taken from the set of non-negative reals;*
- $Steps \subseteq S \times (\mathcal{A}ct \cup \mathbb{T}) \times \mathsf{Dist}(S)$ *is a* probabilistic transition relation*, such that, if* $(s, a, \mu) \in Steps$ *and* $a \in \mathbb{T}$*, then* μ *is a point distribution.*

A *probabilistic transition* $s \xrightarrow{a,\mu} s'$ is made from a state $s \in S$ by first nondeterministically selecting an action-distribution or duration-distribution pair (a, μ) such that $(s, a, \mu) \in Steps$, and second by making a probabilistic choice of target state s' according to the distribution μ, such that $\mu(s')>0$. Duration-distribution pairs always comprise a point distribution, whereas the distributions of action-distribution pairs can be arbitrary.

We consider two ways in which a timed probabilistic system's computation may be represented: using paths and adversaries. A *path* represents a particular resolution of both nondeterminism *and* probability. Formally, a path of a timed probabilistic system is a non-empty finite or infinite sequence of probabilistic transitions

$$\omega = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} s_2 \xrightarrow{a_2,\mu_2} \cdots .$$

We denote by $\omega(i)$ the $(i+1)$th state of ω, $last(\omega)$ the last state of ω if ω is finite and $step(\omega, i)$ the action or duration associated with the i-th transition (that is $step(\omega, i) = a_i$). By abuse of notation, we say that a single state s is a path of length 0. The set of finite (infinite) paths starting in the state s is denoted by $Path_{fin}(s)$ ($Path_{ful}(s)$). For any infinite path $\omega = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} \cdots$, the accumulated duration up to the $(n+1)$-th state of ω is defined by:

$$\mathcal{D}_\omega(n+1) \stackrel{\text{def}}{=} \sum\{|a_i \mid 0 \leq i \leq n \wedge a_i \in \mathbb{T}|\}.$$

In contrast to a path, an *adversary* represents a particular resolution of nondeterminism *only*. More precisely, an adversary is a function which chooses an outgoing distribution in the last state of a path. Formally, we have the following definition.

Definition 9 *Let* $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$ *be a timed probabilistic system. An adversary* A *of* TPS *is a function mapping every finite path* ω *of* TPS *to a pair* (a, μ) *such that* $(last(\omega), a, \mu)$ *is an element of* $Steps$*. For any state* $s \in S$*, let* $Path^A_{fin}(s)$ *and* $Path^A_{ful}(s)$ *denote the subsets of* $Path_{fin}(s)$ *and* $Path_{ful}(s)$ *which correspond to* A*.*

The behaviour of a timed probabilistic system $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$ under a given adversary A is purely probabilistic. More precisely, for a state $s \in S$, the behaviour from state s can be described by the infinite-state DTMC $\mathcal{D}^A_s = (S^A_s, s, \mathbf{P}^A_s)$, where:

- $S^A_s = Path^A_{fin}(s)$;
- for any two finite paths $\omega, \omega' \in S^A_s$:

$$\mathbf{P}^A_s(\omega, \omega') = \begin{cases} \mu(s') & \text{if } \omega' \text{ is of the form } \omega \xrightarrow{a,\mu} s' \\ & \text{and } A(\omega) = (a, \mu) \\ 0 & \text{otherwise.} \end{cases}$$

There is a one-to-one correspondence between the paths of $\mathcal{D}^A$ and the set of paths $Path^A_{ful}(s)$ in the timed probabilistic system. Hence, using the probability measure over DTMCs given in Section 2.3 we can define a probability measure $Prob^A_s$ over the set of paths $Path^A_{ful}(s)$.

We also use *randomized adversaries*, where a randomized adversary B is a function B mapping every finite path ω to a distribution over $\{(a, \mu) \mid (last(\pi), a, \mu) \in Steps\}$. Similarly to the above, we can associate with any randomized adversary a probability measure over the set of paths of the adversary (see, for example, [20, 43]).

We restrict our attention to *time-divergent adversaries*, a common restriction imposed in real-time systems so that unrealisable behaviour (i.e. corresponding to time not advancing beyond a time bound) is disregarded during analysis. We say that an infinite path ω is *divergent* if for any $t \in \mathbb{R}$, there exists $j \in \mathbb{N}$ such that $\mathcal{D}_\omega(j)>t$. It is easy to see that the set of divergent paths is measurable under any adversary identified in the previous paragraph.

Definition 10 *An adversary* A *for a timed probabilistic system* TPS *is* divergent *if and only if, for each state* s *of*

TPS, *the probability* $Prob_s^A$ *assigns to the divergent paths of* $Path_{ful}^A(s)$ *is* 1. *Furthermore, let* Adv_{TPS} *be the set of divergent adversaries of* TPS.

3 Probabilistic Timed Automata

In this section we review the definition of probabilistic timed automata [35], a modelling framework for timed probabilistic systems. The formalism is derived from classical timed automata [1, 2] extended with discrete probability distributions over edges.

3.1 Time, clocks and zones

Let $\mathbb{T} \in \{\mathbb{R}, \mathbb{N}\}$ be the *time domain* of either the non-negative reals or naturals. Let $\mathcal{X}$ be a finite set of variables called *clocks* which take values from the time domain $\mathbb{T}$. A point $v \in \mathbb{T}^{\mathcal{X}}$ is referred to as a *clock valuation*. Let $\mathbf{0} \in \mathbb{T}^{\mathcal{X}}$ be the clock valuation which assigns 0 to all clocks in $\mathcal{X}$. For any $v \in \mathbb{T}^{\mathcal{X}}$ and $t \in \mathbb{T}$, the clock valuation $v \oplus t$ denotes the *time increment* for v and t (we present two alternatives for $\oplus$ in Section 3.3; for the time domain $\mathbb{R}$ it is standard addition $+$). We use $v[X{:=}0]$ to denote the clock valuation obtained from v by resetting all of the clocks in $X \subseteq \mathcal{X}$ to 0, and leaving the values of all other clocks unchanged.

Let $Zones(\mathcal{X})$ be the set of *zones* over $\mathcal{X}$, which are conjunctions of atomic constraints of the form $x \sim c$ for $x \in \mathcal{X}$, $\sim \in \{\leq, =, \geq\}$, and $c \in \mathbb{N}$. The clock valuation v *satisfies* the zone ζ, written $v \triangleleft \zeta$, if and only if ζ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value from v. Readers familiar with timed automata will note that we consider the syntax of *closed, diagonal-free* zones, which *do not* feature atomic constraints of the form $x > c$ or $x < c$ (closed) or $x - y \sim c$ (diagonal free).

3.2 Syntax of probabilistic timed automata

We now introduce formally probabilistic timed automata. Observe that we extend the original definition of [35] with urgent events, a well-established concept for classical timed automata [27, 19].

Definition 11 *A* probabilistic timed automaton PTA *is a tuple of the form* $(L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *where:*

- L *is a finite set of* locations*:*
- $\bar{l} \in L$ *is the* initial location*;*
- $\mathcal{X}$ *is a set of* clocks*;*
- Σ *is a finite set of* events*, of which* $\Sigma_u \subseteq \Sigma$ *are declared as being* urgent*;*
- *the function* $inv : L \to Zones(\mathcal{X})$ *is the* invariant condition*;*
- *the finite set* $prob \subseteq L \times Zones(\mathcal{X}) \times \Sigma \times \mathsf{Dist}(2^{\mathcal{X}} \times L)$ *is the* probabilistic edge relation*.*

Note that we often refer to the model presented above as *closed, diagonal-free probabilistic timed automata*, in order to distinguish the zones used with those in previous work [35].

A *state* of a probabilistic timed automaton is a pair (l, v) where $l \in L$ and $v \in \mathbb{T}^{\mathcal{X}}$ are such that $v \triangleleft inv(l)$. Informally, the behaviour of a probabilistic timed automaton can be understood as follows. The model starts in the initial location $\bar{l}$ with all clocks set to 0, that is, in the state $(\bar{l}, \mathbf{0})$. In this, and any other state (l, v), there is a nondeterministic choice of either (1) making a *discrete transition* or (2) letting *time pass*. In case (1), a discrete transition can be made according to any probabilistic edge $(l, g, \sigma, p) \in prob$ with source location l which is *enabled*; that is, the zone g is satisfied by the current clock valuation v. Then the probability of moving to the location l' and resetting all of the clocks in X to 0 is given by $p(X, l')$. In case (2), the option of letting time pass is available only if the invariant condition $inv(l)$ is satisfied while time elapses and there does not exist an enabled probabilistic edge with an urgent event.

Note that a *timed automaton* [2] is a probabilistic timed automaton for which every probabilistic edge (l, g, σ, p) is such that $p = \mu_{(X,l')}$ (the point distribution assigning probability 1 to (X, l')) for some $(X, l') \in 2^{\mathcal{X}} \times L$.

Higher-level modelling. To aid higher-level modelling, a notion of urgency can be associated with locations, in addition to events. Once an urgent location is entered, it must be left immediately, without time passing. Urgent locations can be represented syntactically in the probabilistic timed automata framework above. Integer variables with bounded ranges, which can be tested within enabling conditions and reset by edge distributions, can also be represented syntactically within the probabilistic timed automaton framework by encoding the values of such variables within locations [45].

It is often useful to define complex systems as the *parallel composition* of a number of interacting subcomponents. The definition of the parallel composition operator $\|$ of probabilistic timed automata uses ideas from the theory of (untimed) probabilistic systems [44] and classical timed automata [2]. Let $\mathsf{PTA}_i = (L_i, \bar{l}_i, \mathcal{X}_i, \Sigma_i, inv_i, prob_i)$ for $i \in \{1, 2\}$ and assume that $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$.

Definition 12 *The* parallel composition *of two probabilistic timed automata* PTA_1 *and* PTA_2 *is the probabilistic timed automaton*

$$\mathsf{PTA}_1 \| \mathsf{PTA}_2 = (L_1 \times L_2, (\bar{l}_1, \bar{l}_2), \mathcal{X}_1 \cup \mathcal{X}_2, \Sigma_1 \cup \Sigma_2, inv, prob)$$

such that

- $a \in \Sigma_1 \cup \Sigma_2$ *is declared as urgent if and only if it is declared urgent in at least one of* PTA_1 *and* PTA_2*;*
- $inv(l, l') = inv_1(l) \wedge inv_2(l')$ *for all* $(l, l') \in L_1 \times L_2$*;*
- $((l_1, l_2), g, \sigma, p) \in prob$ *if and only if one of the following conditions holds:*
  1. $\sigma \in \Sigma_1 \setminus \Sigma_2$ *and there exists* $(l_1, g, \sigma, p_1) \in prob_1$ *such that* $p = p_1 \otimes \mu_{(\emptyset, l_2)}$;

2. $\sigma \in \Sigma_2 \setminus \Sigma_1$ *and there exists* $(l_2, g, \sigma, p_2) \in prob_2$ *such that* $p = \mu_{(\emptyset,l_1)} \otimes p_2$;
3. $\sigma \in \Sigma_1 \cap \Sigma_2$ *and there exists* $(l_i, g_i, \sigma, p_i) \in prob_i$ *for* $i = 1, 2$ *such that* $g = g_1 \wedge g_2$ *and* $p = p_1 \otimes p_2$

where for any $l_1 \in L_1$, $l_2 \in L_2$, $X_1 \subseteq \mathcal{X}_1$ *and* $X_2 \subseteq \mathcal{X}_2$*:*

$$p_1 \otimes p_2(X_1 \cup X_2, (l_1, l_2)) = p_1(X_1, l_1) \cdot p_2(X_2, l_2) \,.$$

Finally, the notion of *committed locations* can help in the modelling of complex systems. As for urgent locations, no time can pass in committed locations. When a component enters a committed location, the next transition of the compound probabilistic timed automaton *must* correspond to a transition from a component which is in a committed location (possibly in synchronization with transitions of other components). This is in contrast to the case of an urgent location, in which the next transition of the compound model need not be one of the outgoing transitions of the urgent location. Like urgent locations, committed locations can be represented syntactically in the probabilistic timed automata framework above.

3.3 Semantics of probabilistic timed automata

We now give the semantics of probabilistic timed automata defined in terms of timed probabilistic systems. Observe that the definition is parameterized by both the time domain $\mathbb{T}$ and the time increment operator $\oplus$. The time increment operator is a binary operator which takes a clock valuation $v \in \mathbb{T}^{\mathcal{X}}$ and a time duration $t \in \mathbb{T}$, and returns a clock valuation $v \oplus t \in \mathbb{T}^{\mathcal{X}}$ which represents, intuitively, the clock valuation obtained from v after t time units have elapsed.

Definition 13 *Let* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *be a probabilistic timed automaton. The* semantics of PTA with respect to the time domain $\mathbb{T}$ and time increment $\oplus$ *is the timed probabilistic system* $[\![\mathsf{PTA}]\!]^{\oplus}_{\mathbb{T}} = (S, \bar{s}, \Sigma, \mathbb{T}, Steps)$ *such that:*

- $S \subseteq L \times \mathbb{T}^{\mathcal{X}}$ *where* $(l, v) \in S$ *if and only if* $v \triangleleft inv(l)$*;*
- $\bar{s} = (\bar{l}, \mathbf{0})$*;*
- $((l, v), a, \mu) \in Steps$ *if and only if one of the following conditions holds:*

  Time transitions. $a = t \in \mathbb{T}$ *and* $\mu = \mu_{(l, v \oplus t)}$ *such that:*

  1. $v \oplus t' \triangleleft inv(l)$ *for all* $0 \leq t' \leq t$*;*
  2. *for all probabilistic edges of the form* $(l, g, \sigma, -) \in prob$*, if* $v \triangleleft g$*, then* $\sigma \notin \Sigma_u$ *(no urgent transitions are enabled);*

  Discrete transitions. $a = \sigma \in \Sigma$ *and there exists* $(l, g, \sigma, p) \in prob$ *such that* $v \triangleleft g$ *and for any* $(l', v') \in S$*:*

$$\mu(l', v') = \sum_{\substack{X \subseteq \mathcal{X} \,\&\, \\ v' = v[X:=0]}} p(X, l') \,.$$

The summation in the definition of discrete transitions is required for the cases in which multiple clock resets result in the same target state.

In our setting, the semantics falls into two classes, depending on whether the underlying model of time is the positive reals or the naturals. If $\mathbb{T} = \mathbb{R}$ we let $\oplus$ equal $+$ (standard addition) and refer to $[\![\mathsf{PTA}]\!]^{+}_{\mathbb{R}}$ as the *dense-time semantics* of the probabilistic timed automaton PTA. In contrast, if $\mathbb{T} = \mathbb{N}$, we let $\oplus$ equal $\oplus_{\mathbb{N}}$ which is defined below, and refer to $[\![\mathsf{PTA}]\!]^{\oplus_{\mathbb{N}}}_{\mathbb{N}}$ as the *integral semantics* of PTA. To define $\oplus_{\mathbb{N}}$, first, for any $x \in \mathcal{X}$, let $\mathbf{k}_x$ denote the greatest constant the clock x is compared to in the zones of PTA. Then, for any clock valuation $v \in \mathbb{N}^{\mathcal{X}}$ and time duration $t \in \mathbb{N}$, let $v \oplus_{\mathbb{N}} t$ be the clock valuation of $\mathcal{X}$ which assigns the value $\min\{v_x + t, \mathbf{k}_x + 1\}$ to all clocks $x \in \mathcal{X}$ (although the operator $\oplus_{\mathbb{N}}$ is dependent on PTA, we elide a sub- or superscript indicating this for clarity).

Note that the definition of integral semantics for probabilistic timed automata is a generalization of the analogous definition for the classical model in [12]. As we always use the same type of time increment for a particular choice of time domain, we henceforth omit the $+$ and $\oplus_{\mathbb{N}}$ superscripts from the notation, and write $[\![\mathsf{PTA}]\!]_{\mathbb{R}}$ and $[\![\mathsf{PTA}]\!]_{\mathbb{N}}$, respectively. The fact that the integral semantics of a probabilistic timed automaton is finite, and the dense-time semantics of probabilistic timed automaton is generally uncountable, can be derived from the definitions.

It is not difficult to check that the semantics of the parallel composition of two probabilistic timed automata corresponds to the semantics of the parallel composition of their individual semantic timed probabilistic systems. Formally, we overload the parallel composition operator $\|$ such that $\mathsf{TPS}_1 \| \mathsf{TPS}_2$ denotes the timed probabilistic system obtained from the parallel composition of the timed probabilistic systems TPS_1 and TPS_2 in the standard manner [44]. Two timed probabilistic systems $\mathsf{TPS}_1 = (S_1, \bar{s}_1, \mathcal{A}ct, \mathbb{T}, Steps_1)$ and $\mathsf{TPS}_2 = (S_2, \bar{s}_2, \mathcal{A}ct, \mathbb{T}, Steps_2)$ are *isomorphic* if there exists a bijection $f : S_1 \to S_2$ such that $(s_1, a, \mu) \in Steps_1$ if and only if $(f(s_1), a, f(\mu)) \in Steps_2$, where $f(\mu) \in \mathsf{Dist}(S_2)$ is the distribution defined by $f(\mu)(s_2) = \mu(f^{-1}(s_2))$ for each $s_2 \in S_2$. For the probabilistic timed automata PTA_1 and PTA_2 with disjoint clock sets, $[\![\mathsf{PTA}_1 \| \mathsf{PTA}_2]\!]_{\mathbb{T}}$ and $[\![\mathsf{PTA}_1]\!]_{\mathbb{T}} \| [\![\mathsf{PTA}_2]\!]_{\mathbb{T}}$ are isomorphic, both for the integral and dense-time semantics.

4 Probabilistic and Expected Reachability

In this section, we consider two performance measures for probabilistic timed automata. The first is *probabilistic reachability*, namely the maximal and minimal probability of reaching, from the initial state, a certain set of target states. For a timed probabilistic system $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$, set $F \subseteq S$ of target states, and adversary $A \in Adv_{\mathsf{TPS}}$, let:

$$p^A_{\bar{s}}(F) \stackrel{\text{def}}{=} Prob^A_{\bar{s}}\{\omega \in Path^A_{ful}(\bar{s}) \mid \exists i \in \mathbb{N} \,.\, \omega(i) \in F\} \,.$$

Definition 14 *The* maximal and minimal reachability probabilities *of reaching the set of states F of the timed probabilistic system* TPS *are defined as follows:*

$$p^{\max}_{\mathsf{TPS}}(F) = \sup_{A \in Adv_{\mathsf{TPS}}} p^A_{\bar{s}}(F) \qquad \textit{and}$$
$$p^{\min}_{\mathsf{TPS}}(F) = \inf_{A \in Adv_{\mathsf{TPS}}} p^A_{\bar{s}}(F)\,.$$

The second measure we consider is *expected reachability*, which allows us to compute the expected cost (or reward) accumulated before reaching a certain set of states. Expected reachability is defined with respect a set $F \subseteq S$ of target states and a cost function mapping state-actions and state-durations pairs to real values (the cost of performing an action or letting a certain amount of time pass in the corresponding state, respectively). This measure corresponds to the expected cost (with respect to the given cost function) of reaching a state in F. More formally, for a timed probabilistic system $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$, cost function $\mathcal{C} : S \times (\mathcal{A}ct \cup \mathbb{T}) \to \mathbb{R}$, set $F \subseteq S$ of target states, and adversary $A \in Adv_{\mathsf{TPS}}$, let $\mathrm{E}^A_{\bar{s}}(cost(\mathcal{C}, F))$ denote the usual expectation with respect to the measure $Prob^A_{\bar{s}}$ over $Path^A_{ful}(\bar{s})$, where for any $\omega \in Path^A_{ful}(\bar{s})$:

$$cost(\mathcal{C}, F)(\omega) \stackrel{\text{def}}{=} \sum_{i=1}^{\min\{j \mid \omega(j) \in F\}} \mathcal{C}(\omega(i-1), step(\omega, i-1))$$

if there exists $j \in \mathbb{N}$ such that $\omega(j) \in F$, and $cost(\mathcal{C}, F)(\omega) \stackrel{\text{def}}{=} \top$ otherwise. The value of $cost(\mathcal{C}, F)(\omega)$ equals the total cost, with respect to the cost function $\mathcal{C}$, accumulated until a state in F is reached along the path ω.

Note that, for simplicity, we define the cost of a path which does not reach F to be $\top$, even though the total cost of the path may not be infinite. Hence, the expected cost of reaching F from s is finite if and only if a state in F is reached from s with probability 1. *Expected time reachability* (the expected time with which a given set of states can be reached) is a special case of expected reachability, corresponding to the case when $\mathcal{C}(s, a) = 0$ for all $s \in S$ and $a \in \mathcal{A}ct$ and $\mathcal{C}(s, t) = t$ for all $s \in S$ and $t \in \mathbb{T}$.

Definition 15 *The* maximal and minimal expected costs *of reaching a set of states F under the cost function $\mathcal{C}$ in the timed probabilistic system* TPS *are defined as follows:*

$$e^{\max}_{\mathsf{TPS}}(\mathcal{C}, F) = \sup_{A \in Adv_{\mathsf{TPS}}} \mathrm{E}^A_{\bar{s}}(cost(\mathcal{C}, F))$$
$$e^{\min}_{\mathsf{TPS}}(\mathcal{C}, F) = \inf_{A \in Adv_{\mathsf{TPS}}} \mathrm{E}^A_{\bar{s}}(cost(\mathcal{C}, F))\,.$$

We note that calculating expected reachability is equivalent to the *stochastic shortest path problem* for Markov decision processes; see for example [11, 21].

At the level of probabilistic timed automata, one can define a cost function using a pair $(c_\Sigma, \mathbf{r})$, where $c_\Sigma : L \times \Sigma \to \mathbb{R}$ is a function assigning the cost, in each location, of executing each event in Σ, and $\mathbf{r} \in \mathbb{R}$ gives the rate at which cost is accumulated as time passes (independent of the current location). The associated cost function $\mathcal{C}_{c_\Sigma, \mathbf{r}}$ is defined as follows, for each $(l, v) \in L \times \mathbb{R}^{\mathcal{X}}$ and $a \in \Sigma \cup \mathbb{T}$:

$$\mathcal{C}_{c_\Sigma, \mathbf{r}}((l, v), a) \stackrel{\text{def}}{=} \begin{cases} c_\Sigma(l, a) & \text{if } a \in \Sigma \\ a \cdot \mathbf{r} & \text{otherwise.} \end{cases}$$

A probabilistic timed automaton equipped with a pair $(c_\Sigma, \mathbf{r})$ is a probabilistic generalisation of uniformly priced timed automata [8]. In the following section we will restrict attention to such cost functions, while in Section 4.2 we will consider more general cost functions where the cost per unit time can vary depending on the current location, which can be considered as a probabilistic extension of linearly priced timed automata [9].

Note that, we have only considered non-negative cost functions ($\mathbb{R}$ is the set of non-negative reals). However, all the results presented also hold for the corresponding non-positive cost functions.

For both probabilistic and expected reachability, we can consider reaching a state satisfying a formula which is a conjunction of propositions identifying locations and zones (that is, clock constraints of the form $x \sim c$ for $x \in \mathcal{X}$, $\sim \in \{\leq, =, \geq\}$ and $c \in \mathbb{N}$). Instead of considering these cases separately, we just note that such reachability problems can be reduced to those referring to locations only by modifying syntactically the probabilistic timed automaton of interest (see [35]). Note that, if all the zones appearing in the PTA and the clock constraints present in the formula are closed and diagonal-free, then all the zones appearing in the modified PTA are also closed and diagonal-free.

In the case of probabilistic reachability the types of properties which can be expressed can be classified as follows:

Probabilistic reachability The system can reach a certain set of states with a given maximal or minimal probability. For example, 'with probability at least 0.999, a data packet is correctly delivered'.

Probabilistic time-bounded reachability The system can reach a certain set of states within a certain time deadline and probability threshold. For example, 'with probability 0.01 or less, a data packet is lost within 5 time units'.

Probabilistic cost-bounded reachability The system can reach a certain set of states within a certain cost and probability bound. For example, 'with probability 0.75 or greater, a data packet is correctly delivered with at most 4 retransmissions'.

Invariance The system does not leave a certain set of states with a given probability. For example, 'with probability 0.875 or greater, the system never aborts'.

Bounded response The system inevitably reaches a certain set of states within a certain time deadline with a given probability. For example, 'with probability 0.99 or greater, a data packet will always be delivered within 5 time units'.

On the other hand, expected time reachability allows us to express, for example, 'the expected time until a data packet is delivered is at most 20ms' and 'the expected time until a packet collision occurs is at least 100 seconds'. In general, expected reachability allows us to validate properties including: 'the expected number of retransmissions before the message is correctly delivered is less than 3', 'the expected number of packets sent before failure is at least 300' and 'the expected number of lost messages within the first 200 seconds is at most 10'.

We illustrate the expected reachability approach using the final property as an example. We would first need to modify the probabilistic timed automaton under study by adding a distinct clock (to represent the elapsed time) and location such that, from all locations, once this clock has reached 200 seconds the only transition is to this new location. The set of target states would then be the set containing only the new location. The cost function would equal 0 on all time transitions and events except the event(s) corresponding to a message being lost, whose cost would be set to 1.

This paper is an abbreviated version of [34].

4.1 Correctness of the Integral Semantics

In this section we show, under the restriction that the probabilistic timed automaton under study is closed and diagonal-free, that probabilistic and expected reachability values are the same in the integral and dense-time semantics. Therefore, for this class of probabilistic timed automaton, it suffices to verify the integral semantic model. As mentioned in the previous section, in the case of expected reachability, we restrict our attention to cost functions defined by pairs $(c_\Sigma, \mathbf{r})$, where $c_\Sigma : L \times \Sigma \to \mathbb{R}$ and $\mathbf{r} \in \mathbb{R}$. Note that, an alternative characterization of the cost functions we consider is those that satisfy the following property:

- $\mathcal{C}(s,t) = \mathcal{C}(s',t)$ for all $s, s' \in S$ and $t \in \mathbb{R}$;
- $\mathcal{C}(s, t+t') = \mathcal{C}(s,t) + \mathcal{C}(s,t')$ for all $s \in S$ and $t, t' \in \mathbb{R}$.

Let $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ be a probabilistic timed automaton. For any set of locations $F \subseteq L$, we denote by $F_\mathbb{T}$ the set of all states of $[\![\mathsf{PTA}]\!]^\oplus_\mathbb{T}$ which correspond to these locations; that is

$$F_\mathbb{T} = \{(l,v) \mid l \in F,\ v \in \mathbb{T}^\mathcal{X} \text{ and } v \triangleleft inv(l)\}\,.$$

In this section we state the following two theorems, namely that minimum reachability probabilities and expected costs agree for the continuous and integral semantics, and likewise for the maximum reachability probabilities and expected costs.

Theorem 16 *For any (closed, diagonal-free) probabilistic timed automaton* PTA *and set of locations $F \subseteq L$:*

$$\begin{aligned} p^{\max}_{[\![\mathsf{PTA}]\!]_\mathbb{R}}(F_\mathbb{R}) &= p^{\max}_{[\![\mathsf{PTA}]\!]_\mathbb{N}}(F_\mathbb{N}) \\ p^{\min}_{[\![\mathsf{PTA}]\!]_\mathbb{R}}(F_\mathbb{R}) &= p^{\min}_{[\![\mathsf{PTA}]\!]_\mathbb{N}}(F_\mathbb{N})\,. \end{aligned}$$

Theorem 17 *For any (closed, diagonal-free) probabilistic timed automaton* PTA*, set of locations $F \subseteq L$ and cost function $\mathcal{C}_{c_\Sigma,\mathbf{r}}$:*

$$\begin{aligned} e^{\max}_{[\![\mathsf{PTA}]\!]_\mathbb{R}}(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_\mathbb{R}) &= e^{\max}_{[\![\mathsf{PTA}]\!]_\mathbb{N}}(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_\mathbb{N}) \\ e^{\min}_{[\![\mathsf{PTA}]\!]_\mathbb{R}}(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_\mathbb{R}) &= e^{\min}_{[\![\mathsf{PTA}]\!]_\mathbb{N}}(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_\mathbb{N})\,. \end{aligned}$$

The above theorems are proved using techniques developed in the classical timed automata case. First consider the following definition and lemma which are taken from [26, 28].

Definition 18 *For any $t \in \mathbb{R}$ and $\varepsilon \in [0,1]$ let:*

$$[t]_\varepsilon = \begin{cases} \lfloor t \rfloor & \text{if } t \le \lfloor t \rfloor + \varepsilon \\ \lceil t \rceil & \text{otherwise.} \end{cases}$$

Note that, from Definition 18, it trivially follows that:

$$[t]_1 \le t \le [t]_0 \quad \text{for all } t \in \mathbb{R}. \tag{1}$$

Lemma 19 *For any $t, t' \in \mathbb{R}$, $c \in \mathbb{N}$ and $\sim \in \{\le, =, \ge\}$, if $t - t' \sim c$ then $[t]_\varepsilon - [t']_\varepsilon \sim c$ for all $\varepsilon \in [0,1]$.*

Next, we introduce the following property on paths of probabilistic timed automata.

Lemma 20 *For any path $\omega = (l_0, v_0) \xrightarrow{a_0,\mu_0} (l_1, v_1) \xrightarrow{a_1,\mu_1} \cdots$, $x \in \mathcal{X}$ and $i \in \mathbb{N}$, there exists $j \le i$ such that $v_i(x) = \mathcal{D}_\omega(i) - \mathcal{D}_\omega(j)$.*

Using the above, we now define the *ε-digitization* of a path [28, 26].

Definition 21 (ε-digitization) *For any path:*

$$\omega = (\bar{l}, \mathbf{0}) \xrightarrow{a_0,\mu_0} (l_1, v_1) \xrightarrow{a_1,\mu_1} \cdots$$

of $[\![\mathsf{PTA}]\!]_\mathbb{R}$, its ε-digitization is the path

$$[\omega]_\varepsilon = (\bar{l}, \mathbf{0}) \xrightarrow{a'_0,\mu_0} (l_1, [v_1]_\varepsilon) \xrightarrow{a'_1,\mu_1} \cdots$$

of $[\![\mathsf{PTA}]\!]_\mathbb{N}$ where for any $i \in \mathbb{N}$ and $x \in \mathcal{X}$:

- $[v_i]_\varepsilon(x) = \min([\mathcal{D}_\omega(i)]_\varepsilon - [\mathcal{D}_\omega(j)]_\varepsilon, \mathbf{k}_x + 1)$ *and $j \le i$ such that $v_i(x) = \mathcal{D}_\omega(i) - \mathcal{D}_\omega(j)$ which exists by Lemma 20;*
- *if $a_i \in \Sigma$, then $a'_i = a_i$;*
- *if $a_i \in \mathbb{R}$, then $a'_i = [\mathcal{D}_\omega(i+1)]_\varepsilon - [\mathcal{D}_\omega(i)]_\varepsilon$.*

The well-definedness of this construction – that is, the fact that $[\omega]_\varepsilon$ is a path of $[\![\mathsf{PTA}]\!]_\mathbb{N}$ – follows from Lemma 19, Lemma 20, and the fact that the zones appearing in PTA are closed and diagonal-free. For example, for any $x \in \mathcal{X}$ and $i \ge 0$ such that $a_i \in \mathbb{R}$, by Definition 21 there exists $j \le i$ such that

$$\min([v_i]_\varepsilon(x) + a'_i, \mathbf{k}_x + 1)$$

$= \min([\mathcal{D}_\omega(i)]_\varepsilon - [\mathcal{D}_\omega(j)]_\varepsilon + a'_i, \mathbf{k}_x{+}1)$
$= \min([\mathcal{D}_\omega(i)]_\varepsilon - [\mathcal{D}_\omega(j)]_\varepsilon + [\mathcal{D}_\omega(i{+}1)]_\varepsilon - [\mathcal{D}_\omega(i)]_\varepsilon, \mathbf{k}_x{+}1)$
by construction
$= \min([\mathcal{D}_\omega(i{+}1)]_\varepsilon - [\mathcal{D}_\omega(j)]_\varepsilon, \mathbf{k}_x{+}1)$
rearranging
$= [v_{i+1}]_\varepsilon(x)$
by Definition 21.

The following lemmas which relate the time and cost of a path with those of its digitization.

Lemma 22 *For any path $\omega \in Path_{ful}(\bar{s})$, $\varepsilon \in [0,1]$ and $i \in \mathbb{N}$: $\mathcal{D}_{[\omega]_\varepsilon}(i) = [\mathcal{D}_\omega(i)]_\varepsilon$.*

Lemma 23 *For any path $\omega \in Path_{ful}(\bar{s})$, set of locations $L' \subseteq L$ and cost function $\mathcal{C}_{c_\Sigma,\mathbf{r}}$ we have:*

$$cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{N}})([\omega]_1) \leq cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{R}})(\omega) \leq cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{N}})([\omega]_0) .$$

We now extend the notion of digitization from paths to adversaries. To simplify the presentation, when considering a fixed adversary $A \in Adv_{[\![\mathsf{PTA}]\!]_\mathbb{R}}$, we suppose that the domain of the mapping $[\cdot]_\varepsilon$ is restricted to the set of paths $Path^A_{ful}(\bar{s})$. Using this interpretation we now extend the notion of digitization to adversaries through the following proposition. Note that we extend the digitization notation $[\cdot]_\varepsilon$ to *sets* of paths: for a set Ω of infinite paths, let $[\Omega]_\varepsilon = \{[\omega]_\varepsilon \mid \omega \in \Omega\}$.

Proposition 24 *For any adversary $A \in Adv_{[\![\mathsf{PTA}]\!]_\mathbb{R}}$ and $\varepsilon \in [0,1]$, there exists a (randomized) adversary $B^\varepsilon \in Adv_{[\![\mathsf{PTA}]\!]_\mathbb{N}}$ such that: $Prob^{B^\varepsilon}_{\bar{s}}(\Pi) = Prob^A_{\bar{s}}([\Pi]^{-1}_\varepsilon)$ for all $\Pi \in \mathcal{F}_{Path^{B^\varepsilon}_{ful}(\bar{s})}$.*

Before we consider Theorem 17, we require the following lemma and proposition. Recall the definition of a measurable function (Definition 2).

Lemma 25 *For any adversary $A \in Adv_{[\![\mathsf{PTA}]\!]_\mathbb{R}}$, the mapping $[\cdot]_\varepsilon$ is a measurable function from $(Path^A_{ful}(\bar{s}), \mathcal{F}_{Path^A_{ful}(\bar{s})})$ to the measurable space induced from the set of paths $\{[\omega]_\varepsilon \mid \omega \in Path^A_{ful}(\bar{s})\}$.*

Proposition 26 *For any set of locations $F \subseteq L$, adversary $A \in Adv_{[\![\mathsf{PTA}]\!]_\mathbb{R}}$ and cost function $\mathcal{C}_{c_\Sigma,\mathbf{r}}$:*

$$\mathrm{E}^{B^1}_{\bar{s}}(cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{N}})) \leq \mathrm{E}^A_s(cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{R}})) \leq \mathrm{E}^{B^0}_{\bar{s}}(cost(\mathcal{C}_{c_\Sigma,\mathbf{r}}, F_{\mathbb{N}})) .$$

4.2 Expected Reachability and Variable Cost Functions

In this section we extend our results on expected reachability to the case when, as in (non-probabilistic) linearly priced timed automata [9], the costs associated with the time spent in locations can vary between locations. More precisely, we consider cost functions of the form $\mathcal{C}_{c_\Sigma,r}$ where:

- $c_\Sigma : L \times \Sigma \to \mathbb{R}$ is a function assigning the cost of executing events;
- $r : L \to \mathbb{R}$ is a function assigning to each location the rate at which costs are accumulated as time passes in that location;
- for any $(l,v) \in L \times \mathbb{R}^\mathcal{X}$ and $a \in \Sigma \cup \mathbb{R}$:

$$\mathcal{C}_{c_\Sigma,r}((l,v),a) \stackrel{\text{def}}{=} \begin{cases} c_\Sigma(l,a) & \text{if } a \in \Sigma \\ a \cdot r(l) & \text{otherwise.} \end{cases}$$

Note that, an alternative characterization of such cost functions is those that satisfying:

$\mathcal{C}(s, t+t') = \mathcal{C}(s,t) + \mathcal{C}(s,t')$ for all $s \in S$ and $t, t' \in \mathbb{R}$.

Using these more general cost functions allows us to calculate performance measures such as:

- the expected time the channel is free before N messages are sent (by setting $r(l)$ to be 1 if location l corresponds to a state in which the channel is free, and 0 otherwise);
- the expected time a sender spends waiting for an acknowledgement (by setting $r(l)$ to be 1 if location l corresponds to a state in which the sender is waiting for an acknowledgement, and 0 otherwise);
- the expected power consumption within the first $T(\in \mathbb{N})$ seconds (by setting $r(l)$ to the power usage (Watts) of the location $l \in L$ and $c_\Sigma(l,\sigma)$ to be the power consumption associated with performing the event σ in location l).

As in the previous section, the proof demonstrating that digital clocks are sufficient for calculating expected reachability properties for such cost functions relies on showing that, for any fixed (dense-time semantic) adversary, there exist integral-time semantic adversaries whose expected costs of reaching a set of target states within n transitions bound that of A. We first define, for any adversary A, a sequence of functions $(e^A_n)_{n\in\mathbb{N}}$, where, for any state s, cost function $\mathcal{C}_{c_\Sigma,r}$ and set of target locations F, $e^A_n(\mathcal{C}_{c_\Sigma,r}, F_\mathbb{T}, s)$ equals the expected cost, under the adversary A, of reaching F from s within n transitions. Since adversaries can choose on the basis of history, we first define e^A_n over paths, then restrict to the case of the initial state (paths of length 0).

Definition 27 *Let $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ be a timed probabilistic automata and $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$ be its semantics for the time domain $\mathbb{T}$. For any subset of target locations $F \subseteq L$, cost function $\mathcal{C}_{c_\Sigma,r}$, adversary $A \in Adv_{\mathsf{TPS}}$ and $\omega \in Path^A_{fin}$, if $last(\omega) = (l,v)$ and $A(\omega) = (a,\mu)$, let: $e^A_0(\mathcal{C}_{c_\Sigma,r}, F_\mathbb{T}, \omega) = 0$ and for any $n \geq 0$: $e^A_{n+1}(\mathcal{C}_{c_\Sigma,r}, F_\mathbb{T}, \omega)$ equals 0 if $s \in F_\mathbb{T}$ and equals*

$$\mathcal{C}_{c_\Sigma,r}(l,a) + \sum_{s' \in S} \mu(s') \cdot e^A_n(\mathcal{C}_{c_\Sigma,r}, F_\mathbb{T}, \omega \xrightarrow{a,\mu} s')$$

otherwise.

Lemma 28 *Let* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *be a timed probabilistic automata and* $\mathsf{TPS} = (S, \bar{s}, \mathcal{A}ct, \mathbb{T}, Steps)$ *be its semantics for the time domain* $\mathbb{T}$*. For any subset of target locations* F*, cost function* $\mathcal{C}_{c_\Sigma,r}$*, adversary* $A \in Adv_{\mathsf{TPS}}$*:* $\langle e_n^A(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{T}}, \bar{s})\rangle_{n\in\mathbb{N}}$ *is a non-decreasing sequence converging to* $e_{\bar{s}}^A(cost(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{T}}))$*.*

Lemma 29 *Let* $\mathsf{PTA} = (L, \bar{l}, \mathcal{X}, \Sigma, inv, prob)$ *be a timed probabilistic automata. For any adversary* $A \in Adv_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}}$*, set of target locations* F *and* $n \in \mathbb{N}$*, there exist adversaries* $B, C \in Adv_{[\![\mathsf{PTA}]\!]_{\mathbb{N}}}$ *such that:*

$$e_n^B(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{N}}, \bar{s}) \leq e_n^A(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{R}}, \bar{s}) \leq e_n^C(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{N}}, \bar{s})\,.$$

Theorem 30 *For any (closed, diagonal-free) probabilistic timed automaton* PTA *(where all probability values are rational), set of locations* $F \subseteq L$ *and (non-negative) cost function* $\mathcal{C}_{c_\Sigma,r}$*, we have:*

$$\begin{aligned} e^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}}(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{R}}) &= e^{\max}_{[\![\mathsf{PTA}]\!]_{\mathbb{N}}}(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{N}}) \\ e^{\min}_{[\![\mathsf{PTA}]\!]_{\mathbb{R}}}(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{R}}) &= e^{\min}_{[\![\mathsf{PTA}]\!]_{\mathbb{N}}}(\mathcal{C}_{c_\Sigma,r}, F_{\mathbb{N}})\,. \end{aligned}$$

4.3 Limitations of Digital Clocks

In this section we investigate the limitations of digital clocks when analysing probabilistic timed automata. In particular, in Section 4.3.1 we show that the integral-time semantics does not preserve probabilistic stopwatch-bounded reachability properties, while in Section 4.3.2 we show that the integral semantics does not preserve the satisfaction of the probabilistic timed temporal logic PTCTL [35].

4.3.1 Probabilistic Stopwatch-Bounded Reachability

We now show, by means of a counter-example, that the integral-time semantics does not preserve probabilistic stopwatch-bounded reachability properties; that is, properties concerned with the probability of reaching a certain set of states before the time spent in a certain set of locations reaches a bound. This means that properties such as 'the probability that a message is correctly delivered while spending at most T time units waiting for an acknowledgement is greater than 0.9' cannot in general be verified correctly using the integral-time semantics.

Consider the probabilistic timed automaton of Figure 1, and suppose that we associate a stopwatch with this automaton which increases at the same rate as real-time ('running') in the locations l_1 and l_3, and remains constant ('stopped') in all other locations. The property we consider is the minimum probability of reaching the location l_4 while the stopwatch (i.e. time spent in l_1 and l_3) remains (less than or) equal to zero. First, under the integral-time semantic model, the transition from $\bar{l}$ can be taken either when the clock x equals 0 or 1.

- If x equals 0 when the transition from $\bar{l}$ is taken, then, on the upper branch, 1 time unit is spent in location l_1 before the location l_4 is reached, while, on the lower branch, 0 time units are spent in location l_3 before the location l_4 is reached.
- If x equals 1 when the transition from $\bar{l}$ is taken, then, on the upper branch, 0 time units are spent in location l_1 before the location l_4 is reached, while, on the lower branch, 1 time unit is spent in location l_3 before the location l_4 is reached.

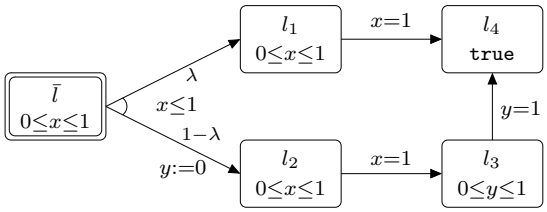


Figure 1. Example demonstrating that probabilistic stopwatch-bounded reachability properties are not preserved.

It then follows that, under the integral-time semantic model, the minimum probability of reaching l_4 while the stopwatch remains equal to zero equals $\min(\lambda, 1-\lambda)$.

On the other hand, for the dense-time semantic model, suppose that the transition from $\bar{l}$ is taken when $x = \delta \in (0, 1)$. Then, on the upper branch, $1 - \delta(> 0)$ time units are spent in location l_1 before the location l_4 is reached, while, on the lower branch, $\delta(> 0)$ time units are spent in location l_3 before the location l_4 is reached. Therefore, for the dense-time semantic model, the minimum probability of reaching location l_4 while the stopwatch remains equal to zero is 0 (consider any adversary which lets some $\delta \in (0, 1)$ time units pass before taking the discrete transition from $\bar{l}$), and hence the minimum probabilities in the integral and dense-time semantic models disagree.

Note that, for the corresponding minimum and maximum expected reachability properties, i.e. the expected time spent in the locations l_1 and l_3 until the location l_4 is reached, the integral and dense-time models agree. More precisely, for the cost function $\mathcal{C}_{c_\Sigma,r}$ such that $c_\Sigma(l, a) = 0$ for all $l \in L$ and $a \in \Sigma$ and $r(l) = 1$ if $l \in \{l_1, l_3\}$ and 0 otherwise, then, for both the integral and dense-time semantics, the minimum and maximum expected cost of reaching location l_4 equal $\min(\lambda, 1-\lambda)$ and $\max(\lambda, 1-\lambda)$ respectively.

4.3.2 The logic PTCTL

PTCTL is a combination of two extensions of the branching temporal logic CTL, the real-time temporal logic TCTL [29] and the probabilistic temporal logic PCTL [25, 14]. The logic TCTL can express timing constraints referring to the clocks of the probabilistic timed automaton and a new set of *formula clocks*, and includes the reset quantifier $z.\phi$, used to reset the formula clock z so that ϕ is

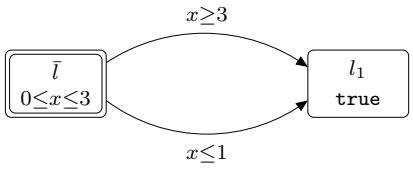


Figure 2. Example demonstrating that the satisfaction of PTCTL formulae are not preserved.

evaluated from a state at which $z = 0$. PTCTL is obtained by enhancing TCTL with the probabilistic operator $\mathcal{P}_{\sim\lambda}[\cdot]$ from PCTL. In the derived logic we can express properties such as 'with probability 0.95 or greater, the value of the system clock x does not exceed 3 before 8 time units have elapsed', which is represented as the PTCTL formula $z.\mathcal{P}_{\geq 0.95}[(x \leq 3)\ \mathcal{U}\ (z=8)]$. For details on the formal syntax and semantics of PTCTL see, for example, [35].

Note that, if we restrict attention to formulae which do not contain nested $\mathcal{P}_{\sim\lambda}[\cdot]$ operators, it is straightforward to extend our results to show that the satisfaction, in the initial state, of such formulae is preserved by the integral semantics (under the restriction that all clock constraints appearing in the formula are closed and diagonal-free). This result follows from the fact that the satisfaction of such a formula reduces to computing either the maximum or minimum reachability probability on a closed diagonal-free probabilistic timed automaton. However, as the following example demonstrates, digital clocks do not suffice for the verification of PTCTL formulae containing *nested* $\mathcal{P}_{\sim\lambda}[\cdot]$ operators.

Consider the (probabilistic) timed automaton given in Figure 2. In the integral-time semantic model, no state of the corresponding timed probabilistic system satisfies the formula $\phi = z.\mathcal{P}_{<1}[\texttt{true}\ \mathcal{U}\ (a_{l_1} \wedge z \leq 1)]$ (for all adversaries the probability of reaching the location l_1 within 1 time unit is less than 1). More precisely, if the automaton is in location $\bar{l}$ and the clock x has an integer value, then there exists an adversary such that (with probability 1) location l_1 is reached within 1 time unit. Therefore, since no state of the integral-time semantic model can reach a state satisfying ϕ, the formula $\mathcal{P}_{<1}[\texttt{true}\ \mathcal{U}\ \phi]$ (for all adversaries the probability of reaching a state satisfying the formula ϕ is less than 1) is trivially true in the initial state $(\bar{l}, \mathbf{0})$.

On the other hand, for the dense-time semantics, when the automaton is in location $\bar{l}$ and the clock x is in the interval $(1, 2)$, then l_1 cannot be reached without letting more than 1 time unit elapse. Hence, starting from the initial state $(\bar{l}, \mathbf{0})$ and letting time pass until $x \in (1, 2)$ the formula ϕ becomes true, and hence $\mathcal{P}_{<1}[\texttt{true}\ \mathcal{U}\ \phi]$ is not satisfied in the initial state.

4.4 Model Checking

To apply model checking methods we must first ensure that the system we consider has only finitely many states and is finitely branching. From the construction, the integral semantics has only finitely many states. However, to ensure finite branching, we must restrict the delays in the integral semantic models from $\mathbb{N}$ to some finite set. For example, we can restrict delays to have duration 1 only and, since any time transition of duration in $\mathbb{N}$ can be modelled by a sequence of time transitions of duration 1 and we restrict attention to divergent adversaries, nothing is lost by omitting delays of duration greater than 1 or of duration 0.

The model checking algorithms for both probabilistic and expected reachability are available in the literature; for probabilistic reachability see [14, 7], and for expected reachability see [20, 21]. In both cases verification reduces to solving a linear optimization problem for which one can apply iterative methods. We also note that, in [20], algorithms for checking for the presence of divergent adversaries are given.

The integral semantic model can suffer from the state space explosion problem; in particular, the size of the models is exponential in the number of clocks and the largest constant that the clocks are compared to. An abstraction technique which can be used to reduce the size of the model under study is that of changing the *time scale*, since this can reduce the constants that clocks are compared to. More formally, one can increase the time unit and then round upper bounds on the values of the constraints up, lower bounds down. For (non-probabilistic) timed automata, it is established in [4] that the trace set of the timed automaton after such a transformation includes that of the original model. It follows that, in the probabilistic setting, carrying out our model-checking on the transformed automaton gives bounds on the performance indices of the original automaton. More precisely, the computed *maximum* probabilistic and expected reachability measures on the transformed model are *upper* bounds and the minimum probabilistic and expected reachability measures are lower bounds on those that would be obtained for the original automaton.

4.4.1 The probabilistic model checker PRISM

PRISM [31, 41] is a probabilistic model checker developed at the University of Birmingham. PRISM is a symbolic model checker which employs MTBDDs, a variant of Binary Decision Diagrams that enables the storage and efficient manipulation of probability matrices. The current implementation of PRISM supports the analysis of *finite*-state probabilistic models of the following three types: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. These models are described in a high-level language, a variant of reactive modules [3] based on guarded commands.

PRISM accepts specifications in probabilistic tempo-

ral logics. This allows us to express various probabilistic properties such as 'event E happens with probability 1', and 'the probability of cost exceeding C is 95%'. The model checker then analyses the model and checks if the property holds in each state. In the case of MDPs, specifications are written in the logic PCTL, and for the analysis PRISM implements the algorithms of [25, 14, 7].

The current release version of PRISM (2.0) supports the verification of probabilistic reachability properties. A prototype extension of PRISM has also been developed which implements the algorithms of [20, 21] for calculating expected costs, and hence enables the computation of expected reachability properties. For further details see [41, 32, 40].

We note that by using integral semantics and PRISM, and hence MTBDDs, we see similar advantages to those reported in [12, 13] for modelling and verifying classical timed automata using integral semantics and BDDs. In particular, by using only MTBDDs we are able to model and verify large and complex probabilistic real-time systems.

4.4.2 Modeling probabilistic timed automata in PRISM

We now explain the techniques used for modelling (the integral semantic models of) probabilistic timed automata as MDPs in PRISM. First, due to the compositionality of the integral semantic model, if the system under study is a parallel composition of a number of probabilistic timed automata, then the integral semantics of each automaton can be modelled by a PRISM module and the system can be defined as the parallel composition of the modules. The only complication in this approach is representing passage of time; this is accomplished by including a distinct action, `time`, and then labelling the transitions of each module which correspond to time passing with this action. Hence, when a time action is executed, all modules must synchronize on this action.

Since, in the integral semantic model, the possible values of any clock x are in the range $\{0, 1, 2, \ldots, \mathbf{k}_x, \mathbf{k}_x + 1\}$, we can model each clock as a bounded integer-valued variable. Furthermore, we can use bounded integer-valued variables to model the locations of an automaton. The possible transition of an automaton are then defined by a guarded command expressed in terms of these variables.

5 Case Study

In this section, we illustrate the utility of the integral-time semantics of probabilistic timed automata by considering a protocol case study, where all experiments were performed using the probabilistic symbolic model checker PRISM. Further details on this and other case studies, including the model checking statistics, can be found on the PRISM web page [41].

5.1 ZeroConf Dynamic configuration protocol for IPv4 link-local addresses

Our case study concerns the ZeroConf dynamic configuration protocol for IPv4 link-local addresses [17], which offers a distributed 'plug-and-play' solution in which IP address configuration is managed by individual devices connected to a local network. This work extends the results presented in [33]. The protocol has also been studied in [16, 46].

The aim of the protocol is to configure an IP address for a device which newly joins the local network. The IP address is then used to facilitate local communication between the devices of the network. Henceforth, we refer to devices which partake of the protocol as *hosts*. When a host connects to the network, it first randomly selects an IP address from a pool of 65024 available addresses (the Internet Assigned Number Authority has allocated the addresses from 169.254.1.0 to 169.254.254.255 for the purpose of such link-local networks). The host waits a random time of between 0 and 2 seconds before starting to send four *Address Resolution Protocol* (ARP) packets, called *probes*, to all of the other hosts of the network. Probes contain the IP address selected by the host, operate as requests to use the address, and are sent at 2 second intervals. A host which is already using the address will respond with an ARP reply packet, asserting its claim to the address, and the original host will restart the protocol by reconfiguring, where reconfiguration involves randomly choosing a new address and sending new probes. Each time a host witnesses an ARP packet with an address which conflicts with the address that it has chosen, a counter is incremented. If the counter reaches the value 10, then the host 'backs off' and remains idle for at least one minute. If the host sends four probes without receiving an ARP reply packet, then it commences to use the chosen IP address. The host also sends confirmation of this fact to the other hosts of the network by means of two further messages, called *gratuitous* ARPs, which are also sent at 2 second intervals. The protocol has an inherent degree of redundancy, for example with regard to the number of repeated ARP packets sent, in order to cope with message loss. Indeed, message loss makes possible the undesirable situation in which two or more hosts use the same IP address simultaneously.

A host which has commenced using an IP address must reply to ARP packets containing the same IP addresses that it receives from other hosts. It continues using the address unless it receives any ARP packet other than a probe (for example, a gratuitous ARP) containing the IP address that it is using currently. In such a case, the host can either *defend* its IP address, or *defer* to the host which sent the conflicting ARP packet. The host may only defend its address if it has not received a previous conflicting ARP packet within the previous ten seconds; otherwise it is forced to defer. A defending host replies by sending an ARP packet, thereby indicating that it is using, and will continue to use, the IP address. A deferring host does not

send a reply; instead, it ceases using its current IP address, and reconfigures its IP address by restarting the protocol.

As in [46], we assume a broadcast-based communication medium with no routers (for example, a single wire), in which messages arrive in the order in which they are sent. In contrast to the analytic analysis of the protocol of Bohnenkamp et al. [16], we model the possibility that a device could surrender an IP address that it is using to another host; and in contrast to timed-automata-based analysis of Zhang and Vaandrager [46], we model some important probabilistic characteristics of the protocol, and consider parameters more faithful to the standard (such as the maximum number of times a device can witness an ARP packet with the same IP address as that which it wishes to use before 'backing off' and remaining idle for at least one minute).

In the standard [17], there is no mention of what a host should do with messages corresponding to its current IP address (i.e. the probes and gratuitous ARP packets specified in the standard) which are in its output buffer (i.e. those that have yet to be sent), when it reconfigures (chooses a new IP address). However, when the host does reconfigure, unless it picks the same IP address, which happens with a very small probability ($1/65024$), these messages are not relevant. In fact, such messages will slow down the network and may even make hosts reconfigure when they do not need to. We therefore considered two different versions of the protocol: one where the host leaves the messages within its output buffer (`NoReset`) and another where the host clears its buffer when it is about to choose a new IP address (`Reset`).

5.1.1 Modelling the dynamic configuration protocol

We consider in detail one *concrete host*, which is attempting to configure an IP address for a network in which there are N *abstract hosts* which have already configured IP addresses. These hosts are called abstract because we do not study their behaviour in depth. When the concrete host picks an address, the probability of this address being *fresh* (not in use by an abstract host) is $(65024-N)/65024$. We also assume that the concrete host never picks the same IP address twice, as this happens with a very small probability. Also, the (continuous) uniform choice over [0,2], made by the concrete host to determine the delay before it sends its first probe, is abstracted to a discrete uniform choice over $\{0, 1, 2\}$.

To enable the analysis of the protocol under different network scenarios we consider two different values of N corresponding to networks with a different numbers of hosts. More precisely, we consider the cases when N=1000 (the value taken in [16]) and N=20.

Due to space limitations, we omit most of the details of modelling. For more detail see [33, 41, 34].

The model for the concrete host is shown in Figure 3. In RECONF, the host chooses a new IP address by moving to the location CHOOSE if it has experienced less than ten address collisions, and to CHOOSEWAIT otherwise. These transitions are labelled with the event *reset* to inform the environment that the host's buffer is to be reset (all messages in its buffer are to be removed). In both CHOOSE and CHOOSEWAIT, the address selection is represented by the assignment $iph := \mathbf{RAND}(1, 2)$, which corresponds to the host randomly selecting an IP address with probability $N/65024$. The assignment to the clock x (a uniform choice between $\{0, 1, 2\}$), which labels the transitions from CHOOSE and CHOOSEWAIT to WAITSP, approximates the random delay of between 0 and 2 made by the host before sending the first probe.

The model for the environment, which consists of the output buffer of the concrete host, in addition to the buffers of all of the abstract hosts, is shown in Figure 4. The dotted box labelled with three transitions which surrounds the model denotes that these transitions are available in *all* of the locations of the model.

5.1.2 Performance Analysis

In this section, we outline our results of using PRISM to verify the integral-time models of the probabilistic timed automata of the dynamic configuration protocol. In the experiments, as explained above we consider the cases when the number of hosts (N) equals 1000 and 20, and vary both the number of probes a host sends (K) and the probability of message loss.

Note that, because we have abstracted certain aspects of the network (for example, the time taken to send a message), the presented results will give upper and lower bounds on the performance of the protocol, for example the actual reachability probability will lie between the minimum and maximum reachability probabilities computed for the model under study.

The **probabilistic reachability** property we consider is the (minimum and maximum) probability of the host using an IP address which is already in use by another host. For both models the results demonstrate that the probabilities decrease as the number of probes increases, which is to be expected: if more probes are sent then there is a greater chance of receiving a reply to a probe when an IP address already in use is chosen (i.e. the chance that not all the probes and responses get lost).

The **time-bounded property** we consider is the (minimum and maximum) probability of the host using a fresh IP address within time T. The results demonstrate that, for small time bounds, the probability of the property is higher when the number of probes K is smaller. This is to be expected since sending more probes takes more time. However, for larger time bounds the probability is larger when more probes are sent. This is due to the fact that, when more probes are sent, there is less chance of using an IP address already in use, and hence not reaching a state where the host uses a fresh IP address.

Expected reachability. We consider the expected cost of a host choosing an IP address and using it. As in [16], the cost is defined as the time to start using an IP address plus an additional cost (E) associated with the host using an address which is already in use. We consider two different values for E, namely 10^6 and 10^{12}. Note that the choice of the value of this additional cost will depend on how damaging it is for two hosts to use the same IP address, which in turn depends on the network and the nature of its devices.

The results for the model `Reset` are presented in Figure 5. Note that, in each graph a log scale has been use to improve readability. The results for the model `NoReset` are similar, although both the minimum and maximum costs are larger for the model `NoReset` (see [41] for further details). This agrees with our intuition, since, as the results for probabilistic reachability demonstrate, when the host does not clear its buffer, there is a greater chance of using an IP address which is already in use, and hence of incurring a greater cost.

These results are similar to those of [16]: as the message loss probability increases, one must increase the number of probes sent in order to reduce the expected cost; however, if too many probes are sent, the expected cost may start to increase. The rationale for this is that, although increasing the number of probes sent decreases the probability of the host using an IP address which is already in use (that is, decreases the chance of incurring the additional cost), it increases the expected time to choose an IP address (because sending more probes takes more time).

Figure 5 also shows that, as the probability of message loss increases, to minimize the expected costs one must send more probes (increase the value of K). Similarly, the results presented demonstrate the fact that, when the cost of using an IP address which is already in use by another host increases, one must send more probes to minimize the expected cost.

6 Conclusions

In this paper, we have presented results demonstrating that digital clocks are sufficient for analysing a large class of probabilistic timed automata and performance properties. Since many of today's protocols include both timing and probabilistic behaviour, this approach is applicable to a wide area, which we demonstrated by analysing the performance of three real-world protocols.

In particular, we have demonstrated that digital clocks are sufficient for the analysis of probabilistic (timed-bounded) reachability and expected reachability against closed, diagonal-free probabilistic timed automata. In the case of expected reachability, these results extend to the cases when the rate of cost accumulation varies in different locations, as in priced or weighted timed automata [9, 5]. Furthermore, we have shown the limitations of this approach: digital clocks are not sufficient for checking stopwatch properties or general PTCTL specifications.

There are still limitations in the size of the models that can be considered using digital clocks. In the case of probabilistic reachability a more efficient approach is to consider the symbolic model checking technique for probabilistic timed automata against PTCTL introduced in [37, 36]. However, in the case of expected reachability, and in particular expected time reachability, it is not clear if there is an alternative, since by using zones the exact timing information may be lost, and hence the best one could hope for would be approximate results. The application of zones to the verification of priced timed automata [38] may be instructive to this line of research.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 18(1):142–157, 1995.

[5] R. Alur, S. Torre, and G. Pappas. Optimal paths in weighted timed automata. In Benedetto and Sangiovanni-Vincentelli [10], pages 49–62.

[6] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In R. de Simone and D. Sangiorgi, editors, *Proc. 9th Int. Conf. Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 470–484. Springer-Verlag, 1998.

[7] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11:125–155, 1998.

[8] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 2001.

[9] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for linearly priced timed automata. In Benedetto and Sangiovanni-Vincentelli [10], pages 147–162.

[10] M. D. Benedetto and A. Sangiovanni-Vincentelli, editors. *Proc. 4th Int. Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[11] D. Bertsekas and J. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

[12] D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In J. Oliveira and P. Zave, editors, *Proc. Symp. Formal Methods Europe (FME'01)*, volume 2021 of *Lecture Notes in Computer Science*, pages 318–343. Springer-Verlag, 2001.

[13] D. Beyer and A. Noack. Efficient verification of timed automata using BDDs. In S. Gnesi and U. Ultes-Nitsche, editors, *Proc. Formal Methods for Industrial Critical Systems (FMICS'01)*, pages 95–113, 2001.

[14] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1995.

[15] P. Billingsley. *Probability and Measure*. John Wiley and Sons: New York, 1979.

[16] H. Bohnenkamp, P. v. d. Stok, H. Hermanns, and F. Vaandrager. Cost-optimisation of the IPv4 zeroconf protocol. In *Proc. Int. Performance and Dependability Symposium (IPDS'03)*, pages 531–540. IEEE Computer Society Press, 2003.

[17] S. Cheshire, B. Adoba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses (draft August 2002). Zeroconf Working Group of the Internet Engineering Task Force (www.zeroconf.org).

[18] C. Daws, M. Kwiatkowska, and G. Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):221–236, 2004.

[19] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. IEEE Real-Time Systems Symposium (RTSS'95)*, pages 66–75. IEEE Computer Society Press, 1995.

[20] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.

[21] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In J. Baeten and S. Mauw, editors, *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1999.

[22] C. Derman. *Finite-State Markovian Decision Processes*. Academic Press: New York, 1970.

[23] A. Gollu, A. Puri, and P. Varaiya. Discretization of timed automata. In *Proc. 33rd IEEE Conf. Decision and Control*, pages 957–958. IEEE Computer Society Press, 1994.

[24] P. Halmos. *Measure Theory*. Springer-Verlag: New York, 1950.

[25] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(4):512–535, 1994.

[26] T. Henzinger. *The Temporal Specification and Verification of Real-time Systems*. PhD thesis, Stanford University, 1991.

[27] T. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In E. Brinksma, R. Cleaveland, and K. Larsen, editors, *Proc. 1st Int. Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.

[28] T. Henzinger, Z. Manna, and A. Puneli. What good are digital clocks? In W. Kuich, editor, *Proc. 19th Int. Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer-Verlag, 1992.

[29] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[30] J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. Springer-Verlag: New York, 2nd edition, 1976.

[31] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer-Verlag, 2002.

[32] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[33] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. In K. Larsen and P. Niebert, editors, *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, 2003.

[34] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Information and Computation*, 2005. Accepted subject to revisions.

[35] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282:101–150, 2002.

[36] M. Kwiatkowska, G. Norman, and J. Sproston. Symbolic computation of maximal probabilistic reachability. In K. Larsen and M. Nielsen, editors, *Proc. 13th Int. Conf. Concurrency Theory (CONCUR'01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 2001.

[37] M. Kwiatkowska, G. Norman, and J. Sproston. Symbolic model checking for probabilistic timed automata. Technical Report CSR-03-10, School of Computer Science, University of Birmingham, 2003.

[38] K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Int. Conf. Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer-Verlag, 2001.

[39] J. Ouaknine. Digitisation and full abstraction for dense-time model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 2002.

[40] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[41] PRISM web page. www.cs.bham.ac.uk/~dxp/prism.

[42] A. Schrijver. *Theory of Linear and Integer Programming*. J. Wiley and Sons: New York, 1986.

[43] R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[44] R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[45] S. Tripakis. *The formal analysis of timed systems in practice*. PhD thesis, Université Joseph Fourier, 1998.

[46] M. Zhang and F. Vaandrager. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using UPPAAL. Technical Report, NIII-R04XX, University of Nijmegen, 2004.

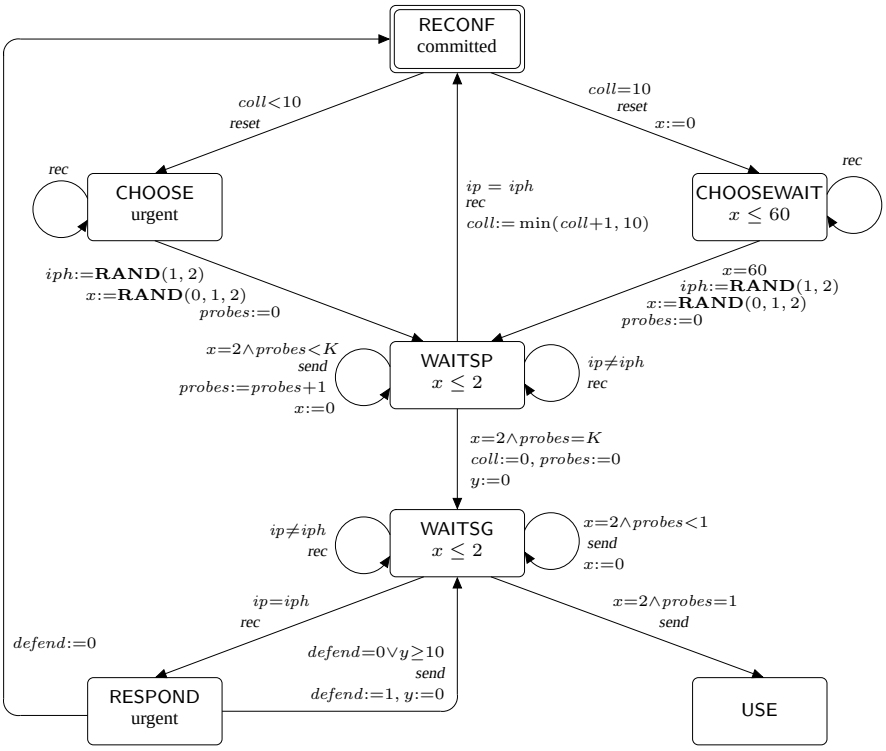


Figure 3. Probabilistic timed automaton for the concrete host (model `Reset`)

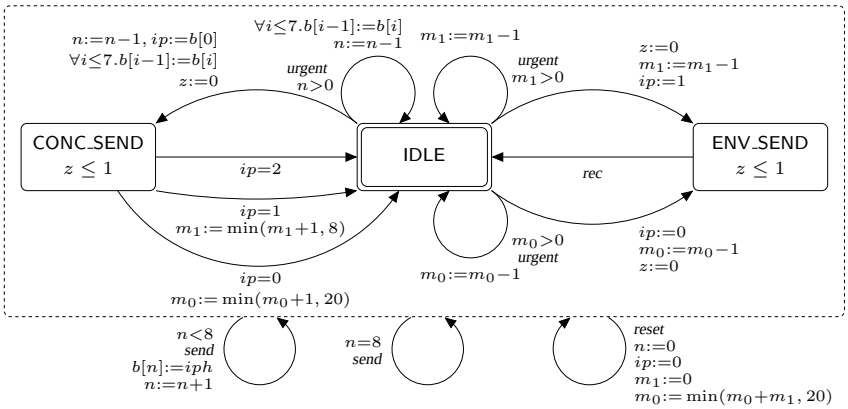
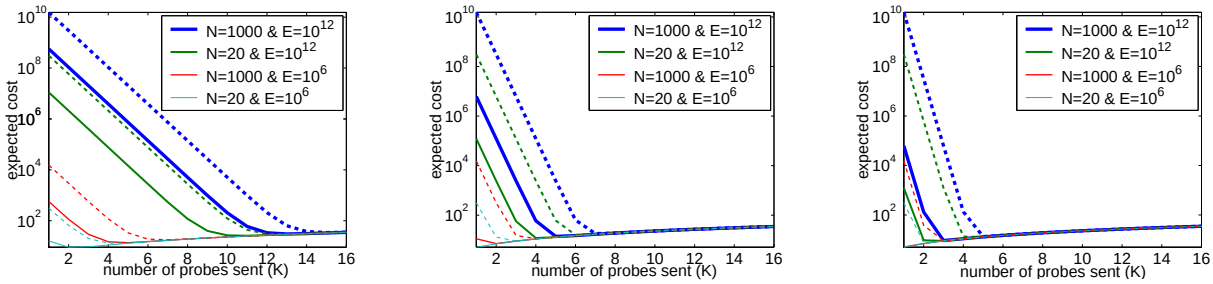


Figure 4. Probabilistic timed automaton for the environment (model `Reset`)



(a) message loss rate 0.1 (b) message loss rate 0.01 (c) message loss rate 0.001

Figure 5. Maximum (dotted) and Minimum (solid) expected cost results

Thème 3

Ordonnancement Temps Réel,
Exécutifs Temps Réel

Les systèmes d'exploitation temps réel

Yvon Trinquet
IRCCyN – UMR CNRS 6597
Ecole Centrale de Nantes
Université de Nantes

Ecole des Mines de Nantes
1, rue de la Nöe
BP 92101 – 44321 – NANTES cedex 3
Yvon.Trinquet@irccyn.ec-nantes.fr

Résumé

Cet article présente quelques concepts des exécutifs temps réel centralisés, logiciels de base permettant d'exécuter sur une machine monoprocesseur un ensemble d'activités coopérantes mais néanmoins concurrentes. Après avoir présenté rapidement l'éventail des réalisations dans ce domaine, l'exposé s'appuie sur un standard du milieu automobile, OSEK-VDX, indépendant de tout éditeur de logiciels. Les services de ce noyau ont été conçus pour pouvoir être utilisés aussi bien dans des domaines tels que le contrôle moteur ou le châssis (très contraints temporellement) que dans un domaine comme l'habitacle ayant moins de contraintes temps réel mais des besoins importants en communication inter-systèmes.

1. Introduction

L'objectif de cet article est de montrer l'intérêt de ces logiciels systèmes que sont les exécutifs temps réel ainsi que la diversité des réalisations, aussi bien dans le domaine commercial, qu'industriel, sans oublier les propositions normatives. C'est d'ailleurs en s'appuyant sur les services d'un standard (non commercial) de l'industrie automobile que sera effectuée la présentation des services de base d'un noyau : OSEK/VDX. Toutes les notions de base des systèmes temps réel sont supposées connues et ne feront donc pas l'objet d'une présentation particulière.

L'analyse du cahier des charges d'une application conduit au recensement des actions élémentaires que doit effectuer le système de pilotage et que l'on regroupe en tâches. Une « tâche » rassemble des fonctions de l'application qui sont exécutées dans les mêmes conditions événementielles. La spécification du comportement, sous forme d'une architecture fonctionnelle (ou logicielle) est naturelle et lisible si on l'exprime sous formes d'actions parallèles, mais ce parallélisme d'expression repose sur l'hypothèse implicite que chaque traitement est réalisé sur un processeur dédié, ce qui ne sera pas forcément vrai à l'exécution. On trouve dans l'architecture des variables d'entrées/sorties représentant des éléments physiques du procédé accessibles via les capteurs/actionneurs (niveau mesuré, commande de vanne …), des liens de coopération entre les tâches ou entre les tâches et l'environnement (liens d'activation, de synchronisation, de communication).

Implanter cette application passe par le choix de la configuration matérielle du système de pilotage sur lequel seront implantées les tâches de l'application. Le choix de la configuration et la sélection des équipements sont naturellement intimement liés, et outre les critères de coût ou de fiabilité, ils dépendent essentiellement des trois facteurs suivants : 1- les contraintes topologiques qui apparaissent lorsque le procédé est une installation éclatée géographiquement ; 2- les contraintes de sûreté de fonctionnement avec l'impératif de maintien du service (nominal ou dégradé) en cas d'anomalie partielle des équipements ; 3- les contraintes temporelles. Ainsi, pour satisfaire une ou plusieurs de ces contraintes, on peut être amené à adopter un système informatique composé de plusieurs stations de traitement interconnectées en réseau., ce qui amène son lot de difficultés nouvelles qui n'apparaissent pas dans le cas monoprocesseur. En effet, les observations de l'état du procédé par des stations distantes peuvent différer en raison de l'asynchronisme du fonctionnement des processeurs et des délais de communication. Dans ce cas, les actions appliquées au système séparément par ces stations peuvent ne pas être cohérentes entre elles. C'est pourquoi l'introduction d'un réseau dans l'architecture matérielle du système de commande est une donnée qui en conditionne également la réalisation logicielle. L'aboutissement de ce développement conduit alors à la définition d'un matériel, d'une décomposition fonctionnelle et logicielle et d'une répartition de ces fonctions sur le matériel, le tout garantissant, ensemble, les différentes contraintes à satisfaire : ce résultat est appelé « *architecture opérationnelle* » de l'application.

L'implantation d'une application est également étroitement conditionnée par un choix majeur : celui de la réalisation du « lien entre les tâches et leurs événements activeurs » qui peut conduire, au niveau

« machine », à deux techniques différentes relevant de deux principes fondamentaux : l'approche « *synchrone* » et l'approche « *asynchrone* ».

La technique de mise en œuvre *synchrone* repose sur deux hypothèses. La première est l'instantanéité des traitements : ils se font en un temps nul. La seconde est la possibilité de percevoir des occurrences d'événements comme étant simultanées car le procédé est observé à des instants discrets. La solution est alors construite en séquentialisant[1] l'exécution des tâches, ce qui évite la « préemption » de toute tâche par une autre. L'approche synchrone a ses inconvénients et ses avantages, le plus marquant des avantages étant la possibilité de faire des vérifications formelles sur les aspects fonctionnels. La démarche synchrone a été à l'origine du développement de langages et environnements de développement pour les systèmes temps réel, tels que Esterel [1] et les Statecharts [2]. L'approche synchrone n'étant pas utilisée dans la mise en œuvre d'une application contrôlée par un exécutif, elle ne sera pas détaillée ici. Le lecteur intéressé pourra se reporter à [3].

La technique de mise en œuvre *asynchrone* consiste à observer en permanence les occurrences d'événement, y compris au cours de l'exécution des tâches, ce qui peut donc engendrer des préemptions. Les dates d'arrivée des événements et leurs ordres relatifs sont connus : la nature de l'action à entreprendre n'est pas ambiguë et son urgence peut être immédiatement prise en compte. Cette démarche tient également compte des temps d'exécution des traitements. Malheureusement, la préemption, outre son coût temporel, engendre de nombreux problèmes : 1- la difficulté de la vérification, les modèles asynchrones étant plus complexes qu'en démarche synchrone ; 2- la mise en œuvre en ligne d'une politique d'ordonnancement ; 3- la protection des ressources partagées. En effet, comme toute tâche est susceptible d'être préemptée à tout moment par d'autres tâches plus prioritaires, les ressources qu'elles se partagent risquent de devenir incohérentes en cas de préemption au milieu de leur manipulation. Il devient nécessaire de garantir cette cohérence par un mécanisme approprié. Tout cela nécessite l'utilisation d'un logiciel spécifique chargé de prendre en compte tous ces besoins, c'est l'exécutif temps réel.

La démarche (asynchrone) qui vient d'être présentée est aussi parfois dénommée « *Event Driven* » ou « *Event Triggered* » car le comportement du système de contrôle est déterminé par l'occurrence des « événements », issus du procédé contrôlé, issus de l'horloge nécessaire pour la perception du temps, issus des tâches elles-mêmes, chaque fois qu'une tâche requiert un service offert par l'exécutif.

Toujours dans le cadre de l'approche asynchrone, on peut concevoir le système de contrôle avec un seul événement externe : celui émanant d'une horloge. Tous les traitements deviennent alors des activités périodiques dont les instants d'activation peuvent être planifiés afin de respecter les contraintes temporelles. Il n'y a plus d'ordonnancement en ligne comme dans le cas précédent mais un ordonnancement pré-calculé hors-ligne. De tels systèmes sont qualifiés de « *Time Driven* » ou « *Time Triggered* », l'exemple par excellence étant le système MARS [4] ou récemment OSEKtime (voir le §4 ci-après). On notera qu'ils présentent une grande « rigidité » conceptuelle par rapport à l'approche « Event Driven ».

La démarche asynchrone, très ancienne, est à l'origine du développement des exécutifs, de nombreux langages temps réel commerciaux et de langages de recherche tels que Real-Time Euclid [5] ou Electre [6]..

2. Différents types d'exécutifs temps réel

Les exécutifs peuvent être sommairement classés en plusieurs catégories [7], [8], [9] :

- les exécutifs « généralistes », pour des modèles d'application variés et incluant de nombreux mécanismes de coopération entre tâches, ouverts sur l'extérieur par des protocoles standards : ce sont en fait les premiers produits apparus qui ont suivi l'évolution des besoins. Ils sont utilisés dans les applications temps réel que l'on peut qualifier de généralistes, présentant des contraintes de temps plus ou moins critiques. On les rencontre dans les systèmes civils et militaires ;
- les exécutifs Unix temps réel. Les applications visées initialement étaient plutôt le marché des applications transactionnelles, mais ces produits introduisent en fait Unix dans les marchés temps réel traditionnels ;
- les exécutifs produits de recherche. Ils mettent l'accent sur certains aspects (ordonnancement, tolérance aux fautes) mais les services de base du noyau sont les mêmes que ceux qui seront présentés dans les catégories correspondant aux produits industriels, parfois pour l'excellente raison qu'ils sont construits au dessus d'un produit commercial. Cet effort de développement a principalement été fait dans les années 90.

Cette classification nous amènera à présenter les services de base (les services du noyau) pour les *exécutifs généralistes*, dans le *cas monoprocesseur*.

[1] La préemption est l'action qui consiste à suspendre l'exécution d'une tâche au profit d'une autre jugée plus prioritaire (critère d'urgence par exemple).

Auparavant on donne quelques indications sur les réalisations « Unix temps réel » ainsi que les exécutifs académiques.

2.1. Unix temps réel

Deux approches ont été suivies pour amener les systèmes Unix vers les applications temps réel : l'approche normative POSIX [10] et celle, plus récente, liée à Linux. L'approche POSIX pour la standardisation des services et interfaces d'un système Unix a conduit à la définition de différentes extensions à l'interface de programmation standard : les extensions temps réel de base (ordonnancement, sémaphores, queues de messages …), les extensions « multi-threads » dans un process et les mutex, principalement. Cette approche a conduit au développement commercial de plusieurs systèmes d'exploitation, avec des caractéristiques temps réel plus ou moins marquées (Unix reste tout de même un « gros » système). Plus récemment l'approche Linux temps réel est liée à l'engouement pour Linux, version « libre » d'Unix vers laquelle des éditeurs de logiciel se tournent depuis plusieurs années. Pour beaucoup d'applications les avantages d'une interface Unix sont supérieurs aux inconvénients dans la mesure où on peut obtenir certaines caractéristiques temps réel. Parmi ces développements on peut citer l'approche RT-Linux [11] ou RTAI [12].

A titre d'exemple, RT-Linux est fondée sur l'idée simple suivante : ne pas chercher à greffer des fonctionnalités temps réel sur un système d'exploitation conçu pour du temps partagé. Ainsi RT-Linux est une tâche de faible priorité s'exécutant sur un noyau temps réel minimal, ce qui permet de minimiser les modifications de Linux, donc de faciliter le suivi lors des évolutions. Les tâches temps réel sont contrôlées par le noyau, elles sont de plus forte priorité que la tâche Linux qui peut donc être préemptée. Ainsi l'application est scindée en deux « domaines » : celui non temps réel avec toutes les ressources de Linux, celui du temps réel avec des ressources limitées. Le domaine temps réel est principalement caractérisé par :

- un ordonnancement préemptif de base à priorité (mais évolutif) ;
- des communication entre les tâches temps réel et les process Linux par FIFO (services non bloquants du côté temps réel) ou par mémoire partagée ;
- jamais de blocage du noyau en attente de ressources Linux ;
- un espace d'adressage partagé avec allocation statique de mémoire.

Une des difficultés est la préemption de Linux avec encore deux approches possibles : celle qui consiste à modifier le noyau Linux pour le rendre préemptible et interruptible, ce qui est difficile car le noyau est complexe et cela complique le suivi des versions ; celle suivi par RT-Linux qui implémente un schéma virtuel d'interruption.

Pour cela on considère que les interruptions sont divisées en deux classes : celles sous le contrôle de RT-Linux et celles sous le contrôle de Linux. Ainsi, lors de son exécution le noyau Linux ne peut pas réellement invalider les interruptions par la macro « cli » (Clear Interrupt): l'occurrence d'une interruption sera traitée si elle est pour RT-Linux, ou sera différée si elle est pour Linux. Lorsque Linux revalide les interruptions (macro « sti » (Set Interrupt)) les interruptions en attente sont traitées.

2.2. Exemples d'exécutifs temps réel académiques

2.2.1. MARS : Maintainable Real-Time Systems

L'approche MARS [4] a été menée principalement à l'Université de Vienne en Autriche par l'équipe de Hermann Kopetz. C'est l'exemple typique d'architecture « Time Triggered ». Les objectifs poursuivis ont été la garantie des temps de réponse dans un système distribué, même en cas de charge maximale et la sûreté de fonctionnement. L'architecture matérielle support, conçue à partir de matériel spécifique, se compose d'un ensemble de groupes (ou grappes, clusters) faiblement couplés. Un groupe est un ensemble d'unités tolérantes aux fautes (FTU) fortement couplées sur un bus temps réel synchrone : le bus MARS. Chaque FTU est composé de 3 unités de traitement (SRU : Single Replication Unit), en parallèle sur le bus, pour traiter les défaillances. Chaque SRU est à arrêt sur défaillance[2] ; elle est conçue en deux parties communicantes : l'unité de communication qui exécute le système d'exploitation et donne à la SRU la propriété d'arrêt sur défaillance, l'unité d'application qui exécute les tâches applicatives [13].

L'architecture logicielle de MARS utilise un modèle qualifié de « transactionnel » pour décrire les activités temps réel : une transaction se décompose en un ensemble de tâches élémentaires réparties sur les unités et dont les exécutions sont contraintes par un graphe de précédence acyclique. Les tâches à contraintes strictes sont périodiques mais il peut y avoir des tâches à contraintes relatives exécutées durant les temps laissés libres. L'ordonnancement est statique, non préemptif et pré-calculé hors-ligne dans le pire des cas prévu par les hypothèses du cahier des charges, ceci pour chaque mode (mode d'initialisation, dégradé, arrêt d'urgence …). Toute l'exécution est « conduite par le temps » (il n'y a pas d'interruption dans MARS autre que l'horloge), une horloge globale très performante synchronisant toutes les unités de traitement afin que chacune travaille dans le quantum de temps pré-alloué. Les variables d'un groupe sont globales et accessibles à

[2] Arrêt sur défaillance : l'unité défaillante s'arrête et n'émet plus aucun message, elle reste silencieuse.

toutes les tâches : elles utilisent le modèle du tableau noir (pas de file, c'est la valeur la plus récente qui est accessible) et véhiculent aussi bien des données que des informations d'état. La transmission des données sur le réseau se fait suivant un protocole TDMA (Time Division Multiple access) afin d'avoir un comportement temporel prédictible et indépendant de la charge : les créneaux temporels (slots) sont pré-alloués aux tâches émettrices par l'ordonnancement hors-ligne.

La tolérance aux fautes est basée sur les unités à arrêt sur défaillance (SRU) en duplication active pour former les FTUs (2 SRU sont en réplication active, la troisième sert à la reconfiguration en cas de défaillance de l'une des deux). Le réseau est, quant à lui, par hypothèse à défaillance par omission[3]. Les états des SRU (hors défaillance) sont toujours cohérents, chaque SRU produisant les mêmes résultats et ayant ses propres créneaux temporels : il n'y a pas besoin de mécanisme de vote ni d'accord (pas de défaillance par valeur). De plus à chaque SRU peuvent être alloués plusieurs créneaux pour le même message afin de les dupliquer dans le temps : ceci participe aussi à la tolérance aux fautes.

MARS se caractérise par une prédictibilité totale, des temps de réponse garantis par conception dans le pire des cas, une tolérance aux fautes élevée et un arrêt du système de manière prédéterminée en cas de défaillance irrécupérable. En contrepartie, la conception d'une application fait forcément preuve d'une grande rigidité, ce qui interdit simplement toute modification ou adaptation car il y a peu de degrés de liberté. Ainsi, l'adjonction ou la modification un peu importante d'une tâche peut nécessiter l'ajout d'une FTU pour garantir les contraintes. Ces développements ont été à l'origine des travaux sur le protocole TTP [14] ainsi que l'architecture TTA (Time Triggered Architecture) qui ont été les développements majeurs dans cette approche.

2.2.2. Le noyau SPRING

Le Spring Kernel [15] a été développé par Jhon Stankovic, Kriti Ramamritham et leur équipe à l'Université du Massachussetts. Les objectifs poursuivis étaient la recherche de la prédictibilité et des garanties temporelles pour certaines tâches d'un système, la prise en compte des tâches apériodiques avec partage de ressources et l'étude d'algorithmes d'ordonnancement réparti, notamment avec coopération dynamique entre les nœuds.

L'architecture matérielle support d'exécution est constituée par un ensemble de nœuds multiprocesseurs en réseau (matériel standard). Un nœud multiprocesseurs comprend : - un (ou plus) processeur d'application pour les tâches applicatives, - un (ou plus) processeur système qui exécute le noyau pour que les « overheads » système ne perturbent pas l'exécution des tâches qui ont une garantie de respect de leurs échéances, - un sous-système d'entrées/sorties (utilisant par exemple l'exécutif VRTX) qui traite les périphériques locaux et la gestion du réseau et notamment toutes les interruptions de façon à ne pas perturber les tâches applicatives par les surcoûts temporels de leur prise en compte.

L'architecture logicielle utilise un modèle d'application complexe : une tâche peut être périodique ou non, à préemption autorisée ou non, possède des contraintes de temps, des contraintes de précédence et de communication. Plusieurs types de tâches ont été identifiés :

- les tâches « critiques », dont le respect des échéances doit être garanti a priori, même en présence de fautes (bien identifiées). Les ressources nécessaires doivent leur être pré-allouées avant exécution. Ces tâches sont en principe peu nombreuses dans les applications visées ;
- les tâches « essentielles » caractérisées elles aussi par une échéance, mais c'est une contrainte qui peut être violée. Il n'est pas possible, d'une manière générale, de leur pré-allouer les ressources nécessaires à leurs exécutions, à cause du caractère très dynamique de ces tâches. Le système (l'ordonnanceur) s'efforce de leur fournir une garantie en ligne pour le respect de l'échéance ;
- les tâches « non-essentielles » pour lesquelles on utilise les temps creux laissés libres par les deux autres catégories.

Un effort important d'études à été mené pour l'ordonnancement des tâches apériodiques avec notamment l'étude d'algorithmes de partage de charge combinant un ordonnanceur local et un ordonnanceur global. Ce dernier, en cas d'impossibilité de garantie locale recherche un site d'accueil pour l'exécution de la tâche (migration du contrôle, pas du code). Par rapport à MARS, il y a ici une combinaison des techniques hors-ligne et en-ligne, car dans les domaines d'applications visés (entre autres les bases de données importantes), on ne peut avoir de connaissance a priori sur l'identité des ressources nécessaires à certaines tâches avant la requête correspondante. Il y a perte de prédictibilité mais beaucoup plus de flexibilité.

2.2.3. ARTS : Advanced Real-time Technology operating System

Ce projet [16] a été développé à l'université Carnegie Mellon au sein de l'équipe de J.P. Lehoczky. Son objectif était de vérifier des technologies logicielles avancées (principalement d'ordonnancement) et créer un environnement d'exécution temps réel distribué qui soit prédictible, analysable et fiable.

L'architecture matérielle est basée sur un réseau de machines conventionnelles (SUN) reliées par un anneau à jeton.

[3] Défaillance par omission : temporairement, le message a émettre n'est pas émis. Il subit un retard infini.

L'architecture logicielle est organisée autour de threads périodiques ou apériodiques, avec des contraintes strictes ou relatives. Les principaux attributs temporels d'un thread sont : la période éventuelle, le temps d'exécution au pire cas, les temps de blocage maximum et une fonction de valeur qui représente l'importance du thread dans l'application. L'ordonnanceur n'est pas « immergé » dans le noyau mais il y a une séparation stricte entre l'algorithme d'ordonnancement et le mécanisme de plus bas niveau de gestion des threads (blocage et dispatching). De ce fait plusieurs stratégies d'ordonnancement ont été mises en place et utilisées à des fins expérimentales (9 sont évoquées : earliest deadline, rate monotonic, fixed priority …).

Un effort important a été également fourni pour développer des outils d'analyse et de vérification, notamment : un analyseur d'ordonnançabilité pour prouver le respect des échéances des tâches à contraintes strictes, les résultats d'ordonnancement des tâches à contraintes relatives selon les différentes stratégies, ainsi que le traitement des surcharges. Ces prédictions sont éprouvées par l'analyse en ligne effectuée avec un moniteur intrusif.

Ce sont les travaux de ce projet qui ont permis de développer les protocoles à héritage de priorité et priorité plafond [17] (d'après nos collègues américains, car Claude Kaiser (CNAM Paris) avait déjà proposé des solutions en 1982 !), l'étude de la résorption des surcharges par l'usage de fonctions de valeur pour les tâches périodiques [18].

3. Services et objets du noyau d'un exécutif temps réel généraliste

Au delà de son rôle premier d'ordonnancer les exécutions des tâches, et de celui de protéger l'accès aux ressources partagées, l'exécutif joue donc un rôle centralisateur, un véritable rôle d'interface qui aiguille les événements reçus du procédé vers les tâches qui les attendent, déclenche le réveil des tâches en attente d'un délai ou d'une heure de démarrage, reçoit et retransmet des signaux de synchronisation ou des données entre des tâches asynchrones. L'exécutif offre ainsi des services accessibles directement par invocation dans les tâches. Ces services sont de différentes natures :

- la *gestion des tâches* : l'exécutif réalise tous les services qui contrôlent l'exécution des tâches comme leur activation, leur suspension, leur reprise, leur terminaison forcée ;
- la *gestion des événements* matériels (interruption) et de synchronisation : la synchronisation est réalisée en permettant aux tâches d'émettre ou de recevoir des signaux « internes ». L'exécutif offre pour cela des services de génération et d'attente de ces « événements logiciels » ;
- la *communication* de messages entre tâches : par boîtes aux lettres, via des ports d'échanges de données, ou sous forme d'appels du type client-serveur ;
- la *gestion du temps* : il s'agit essentiellement de permettre le réveil différé, daté ou périodique des tâches qui le sollicitent ;
- la gestion des *ressources partagées*, de gestion de la *mémoire*, de gestion des *interruptions*, des *exceptions* etc.

La figure 1 représente l'architecture résultante. On y retrouve le rôle de centralisateur que joue l'exécutif par rapport à l'application et l'environnement, et on y a illustré certaines de ses « agences » (entités regroupant l'ensemble des primitives manipulant les mêmes types d'objets).

Pour éviter de présenter les concepts en s'appuyant sur des exemples issus de produits commerciaux, nous allons utiliser le standard OSEK/VDX qui est la référence pour les applications embarquées dans le secteur automobile.

3.1. Historique de la proposition OSEK/VDX

OSEK/VDX[4] est une proposition récente d'exécutif pour l'électronique embarquée dans les automobiles. Cette proposition est le fruit des travaux de nombreux constructeurs automobiles et équipementiers européens depuis septembre 1995. OSEK a été étudié par les équipes principalement allemandes et VDX par le GIE PSA-Renault ; OSEK/VDX est le résultat commun des travaux. L'un des objectifs visés par le consortium est de standardiser les interfaces du système d'exploitation afin de faciliter la fourniture de logiciels par de multiples sources, la portabilité, l'interopérabilité et la réutilisation. Différents travaux ont été menés dans le cadre d'OSEK/VDX :

- OSEK/VDX OS : les services de base du noyau du système d'exploitation ;
- OSEK/VDX COM : les services pour la communication entre des nœuds d'un système réparti ou la communication locale ;
- OSEK/VDX NM (Network Management) : les services de gestion et de surveillance du réseau ;
- OSEK/VDX OIL (OSEK Implementation Language) : langage de description pour la mise en œuvre automatisée d'une application.

L'ensemble des documents de spécification peut être trouvé sur le site [19]. La présentation ci-après porte uniquement sur le noyau du système d'exploitation, le

[4] OSEK/VDX : (Offene Systeme und deren Schnittstellen für dir Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive)

système de communication étant très élaboré et pouvant être un article à part entière.

Tous les objets de la spécification OSEK/VDX sont des objets « statiques », ce qui est naturel dans ce cadre d'application. Une application peut être qualifiée de statique si la durée de vie des objets qui la composent est égale à celle de l'application : les objets sont créés avant le lancement de l'application et ne sont jamais détruits. Par opposition, des objets sont dits « dynamiques » lorsqu'ils peuvent être créés et détruits à tout moment dans la vie de l'application.

3.2. La gestion des tâches

La tâche est l'agent actif de l'application. C'est une portion de code séquentiel correspondant souvent à la notion de procédure (ou fonction) dans le langage de programmation utilisé pour coder l'application. La tâche étant un agent actif, il est naturel de lui associer un diagramme d'état qui représente l'ensemble de ses états possibles ainsi que les conditions associées aux transitions d'états (voir figure 2). OSEK/VDX utilise deux types de tâches définis ci-après :

- les tâches basiques sont des modules sans point bloquant, c'est à dire sans appel de service pouvant provoquer une mise en attente de la tâche. La tâche est activée, elle s'exécute puis elle doit se terminer, les points de synchronisation sont donc seulement au début et à la fin de la tâche. Elle ne possède que trois états : suspended (inactive), ready (attente du processeur), running (elle a le processeur).
- les tâches étendues sont composées de un ou plusieurs modules séparés par des invocations de services éventuellement bloquants (WaitEvent). Le diagramme d'état possède donc un état supplémentaire : l'état waiting.

Selon la classe de conformité de l'exécutif (cf. §3.8.) il est possible ou non d'autoriser la prise en compte de requêtes d'activation d'une tâche (ActivateTask, ChainTask) alors qu'elle est active. Chaque requête est alors mise en attente (pas d'instances multiples) pour être prise en compte quand la tâche se termine (réactivation à la première instruction). Voici quelques exemples de services :

- ActivateTask (<TaskName>) : activation de la tâche désignée ;
- TerminateTask (void): termine la tâche appelante ;
- ChainTask (<TaskName>) : termine la tâche appelante et active la tâche désignée ;
- Schedule (void) : permet, si une tâche plus prioritaire est présente de lui donner le processeur. Ce service n'a pas d'intérêt (et pas d'effet) dans un contexte d'ordonnancement entièrement préemptif.

Remarque : Une application temps réel comporte souvent des tâches périodiques. Nous verrons au §3.5. les mécanismes permettant de gérer l'activation de telles tâches.

Ordonnancement.

Les tâches possèdent une priorité utilisée pour l'ordonnancement. La valeur de priorité est statique, non modifiable sauf par l'exécutif lorsqu'il met en œuvre l'algorithme « Priority Ceiling Protocol » (voir §3.4) pour la gestion des ressources. Plusieurs tâches peuvent être sur le même niveau de priorité selon la classe de conformité (voir §3.8.) mais les techniques d'ordonnancement utilisables sont indépendantes de la classe de conformité et du type de tâche. L'ordonnancement peut être :

- non-préemptif : la tâche rend le processeur lorsqu'elle se termine (TerminateTask), lorsqu'elle active une autre tâche (ChainTask), lorsqu'elle demande un ré-ordonnancement (Schedule) ou lorsqu'elle entre dans un état d'attente (WaitEvent). Dans ce cas le contexte à sauver pour la tâche est minimal ;
- préemptif : toute tâche réveillée et plus prioritaire que la tâche en cours prend le processeur. Le module d'ordonnancement étant considéré comme une ressource occupée ou relâchée, une tâche peut se l'approprier (voir §3.4.) dans un contexte préemptif pour passer en mode non-préemptif ;
- mixte : les deux modes d'ordonnancement sont utilisés en même temps pour des tâches différentes. Dans ce cas chaque tâche possède un attribut indiquant son mode d'ordonnancement. La possibilité d'avoir des tâches ordonnancées selon un mode non préemptif dans un contexte préemptif est utile lorsque des tâches sont courtes (temps d'exécution voisin du temps de changement de contexte), si l'espace mémoire RAM doit être économisé ou encore si la non-préemption est nécessaire.

3.3. La synchronisation des tâches

On distingue en général dans les exécutifs généralistes deux types de synchronisation entre tâches ou entre les tâches et l'environnement : la signalisation synchrone et la signalisation asynchrone. La signalisation asynchrone qui est utilisée, par exemple dans les exécutifs UNIX temps réel pour traiter certains types d'exception, n'a pas été retenue dans OSEK/VDX ; elle ne sera donc pas présentée ici, compte tenu de la place disponible.

Pour OSEK/VDX, la synchronisation est basée sur le mécanisme des événements privés car appartenant au consommateur. Les services liés aux événements ne peuvent être utilisés que par les tâches de type étendu. La synchronisation est de type synchrone, la

tâche réceptrice se mettant explicitement en attente de l'occurrence pour pouvoir être réveillée. Chaque tâche peut posséder un certain nombre d'événements pour lesquels des occurrences seront signalées par d'autres tâches (de type basique ou étendu) ou des ISRs[5]. Seule la tâche propriétaire peut se mettre en attente (OU implicite sur la liste des événements nommés), l'attente n'étant pas automatiquement surveillée temporellement (pas de délai de garde) et l'effacement de l'occurrence étant à la charge de la tâche réceptrice. Voici quelques exemples de services utilisables par les tâches :

- SetEvent (<EventName>,<TaskName>): signale l'occurrence de l'événement nommé pour la tâche désignée ;
- WaitEvent (<EventMask>): met la tâche appelante en attente sur le(s) événement(s) nommé(s) ;
- ClearEvent (<EventMask>) : efface le(s) événement(s) nommé(s) (de la tâche appelante) ;
- GetEvent (<TaskName>,<Events>) : donne l'état des événements de la tâche nommée (permet de savoir, en sortie d'une attente, l'événement arrivé).

La figure 3 montre un exemple de signalisation entre tâches et entre une tâche et l'environnement utilisant des événements privés.

3.4. Partage de ressources et exclusion mutuelle

Il est naturel, dans un contexte où plusieurs tâches coopérantes sont en concurrence, d'avoir à contrôler et obtenir l'accès cohérent à des ressources partagées par les tâches. Les ressources peuvent être de natures très diverses, comme par exemple un système de fichiers accessible en lecture par plusieurs tâches, ou encore une imprimante accessible en exclusion mutuelle par les tâches, ou encore un élément physique du procédé contrôlé qui ne peut être utilisé que par une tâche à la fois, ou tout simplement des données communes. On se trouve donc confronté à un problème de synchronisation entre tâches afin de respecter le protocole d'accès aux ressources, la politique la plus classique étant celle de l'accès en exclusion mutuelle pour une ressource non partageable. Plusieurs techniques sont utilisables pour protéger une section critique, par exemple :

- le blocage des interruptions : ceci n'est envisageable que pour une section très courte et n'est pas toujours possible) ;
- le blocage de la préemption pour la tâche courante (blocage de l'ordonnanceur) qui peut être utilisé pour de courtes sections critiques ;
- l'usage d'objets spécialisés comme les sémaphores ou encore des protocoles plus élaborés comme c'est le cas dans OSEK/VDX.

OSEK/VDX assure la gestion de l'accès concurrent aux ressources partagées par le protocole PCP (« Priority Ceiling Protocol ») [17], ce qui garantit la non-inversion de priorité et l'absence de blocage par usage des services GetResource et ReleaseResource « encadrant » la section critique, sous réserve d'une gestion bien ordonnée (LIFO) des prises de ressources multiples. A l'intérieur de la section critique, les restrictions classiques sur l'appel des services concernent les services de terminaison ou de blocage de la tâche.

Pour la mise en œuvre du protocole PCP une priorité plafond est affectée à chaque ressource. Cette priorité doit être supérieure ou égale à la plus haute priorité des tâches utilisant la ressource tout en étant inférieure à la plus basse priorité des tâches n'utilisant pas la ressource, ces dernières étant néanmoins de plus forte priorité que les tâches utilisant la ressource. Ce protocole est également utilisé pour le partage de ressources entre les tâches et les ISRs ou entre les ISRs ; pour ce faire une priorité virtuelle est allouée à chaque interruption. La ressource système prédéfinie Res_Scheduler permet à une tâche de s'allouer le processeur, puisque l'ordonnanceur est traité comme une ressource. Ceci permet donc de passer en mode non-préemptif.

Deux services permettent de contrôler l'accès aux ressources :

- GetResource (<Res_Name>) : demande l'accès à la ressource désignée avant d'exécuter le code d'utilisation de la ressource. Il est possible d'utiliser plusieurs ressources (différentes) pour une même tâche, pourvu que les appels soient convenablement imbriqués (LIFO) ;
- ReleaseResource (<Res_Name>) : libère l'accès à la ressource désignée.

3.5. Les objets Alarme et Compteur

Ces objets, propres à la spécification OSEK/VDX, permettent principalement le traitement de phénomènes récurrents dans le temps en provenance de l'environnement extérieur, comme par exemple une horloge ou des signaux en provenance d'organes mécaniques d'un moteur automobile (arbre à cames, vilebrequin). Ils constituent des compléments des mécanismes de signalisation par événement. Ils permettent également la mise en œuvre des chiens de garde, par exemple sur l'émission ou la réception des messages. C'est un mécanisme à deux « étages » pour

[5] ISR : Interrupt Service Routine, la routine chargée du traitement de l'interruption. Elle fait, dans ce contexte, le relais entre le mécanisme matériel d'interruption et le mécanisme logiciel de signalisation.

lequel deux objets sont associés : les compteurs, qui ne font pas partie de l'API OSEK mais du langage OIL, et les alarmes.

Un compteur est un objet destiné à l'enregistrement de « ticks » en provenance d'une horloge ou d'un dispositif quelconque émettant des stimulis. C'est un dispositif de comptage ayant une certaine dynamique, qui repasse à zéro après avoir atteint sa valeur maximale (valeur définie à la génération de l'application). Il compte les ticks après une éventuelle pré-division (par exemple 10 ticks représentent une unité pour le compteur). Plusieurs alarmes peuvent être associées à un même compteur, ce qui permet de constituer facilement, par exemple, des bases de temps.

Une alarme est associée (statiquement à la génération du système) à un compteur et une tâche. L'action associée, lors de l'occurrence de l'alarme, peut être :

- l'activation de la tâche ;
- la signalisation d'une occurrence pour un événement de la tâche ;
- l'activation d'une routine pour faire un traitement spécifique (Alarm Callback routine). Elle s'exécute avec certaines restrictions puisqu 'elle est dans le contexte de l'exécutif (ce n'est pas une tâche).

Une alarme peut être unique ou cyclique, absolue ou relative. Si elle est relative, la valeur spécifiée par un paramètre du service est un incrément par rapport à la valeur courante du compteur (expression d'un délai de garde par exemple) ; si elle est absolue, la valeur spécifiée par un paramètre du service définit la valeur du compteur qui active l'alarme. Une autre valeur est spécifiée dans le cas d'une alarme cyclique afin de préciser (en nombre de ticks) la valeur du cycle. Ainsi on sait simplement, sur un compteur lié à l'horloge temps réel, définir au travers de plusieurs alarmes, des tâches périodiques de périodes différentes. La figure 4 montre des exemples d'alarmes unique ou cyclique à partir d'un compteur de dynamique 8 ticks.

Voici des exemples de services :

-SetRelAlarm (<AlarmName>, <Increment>, <Cycle>) : arme l'alarme désignée avec une valeur relative, éventuellement cyclique. Si l'alarme est cyclique la valeur relative indique alors la « distance » par rapport à la première occurrence ;

- SetAbsAlarm (<AlarmName>,<Start>,<Cycle>) : arme l'alarme désignée avec une valeur absolue (référence à une valeur particulière du compteur), éventuellement cyclique. Si l'alarme est cyclique, la valeur relative indique alors la « distance » par rapport à la première occurrence ;
- GetAlarm (<AlarmName>,<Ticks>) : permet d'obtenir pour l'alarme désignée le nombre de ticks restant avant l'occurrence de l'alarme ;
- CancelAlarm (<AlarmName>) : arrête l'alarme désignée (la désactive).

3.6. La communication

Les services de l'API pour la communication sont proposés dans la spécification OSEK/VDX COM, elle ne sera que brièvement évoquée ici. La communication est construite autour des objets « message ». Ceux-ci sont définis à la génération du système quant à leur structure. Deux types de messages sont utilisés :

- ceux utilisant le modèle du tableau noir (Unqueued Message, tampon à une place, le message lu est toujours le dernier reçu) ;
- ceux utilisant une file FIFO (Queued Message), boîte aux lettres utilisant une file de messages, mais contrairement aux boîtes aux lettres classiques il n'y a pas de file de tâches. Il ne peut donc y avoir qu'une seule tâche en attente d'un message.

Le modèle retenu s'appuie sur la communication asynchrone : l'émetteur n'est pas bloqué pendant la transmission, le récepteur n'est pas bloqué s'il n'y a pas de message disponible. Afin d'assurer la « re-synchronisation » sur l'état final de la communication, quatre techniques peuvent être utilisées :

- la surveillance (périodique) sur l'état d'avancement de l'opération ;
- l'activation d'une tâche sur fin d'envoi ou réception d'un message ;
- la signalisation d'une occurrence d'événement sur fin d'envoi ou réception d'un message ;
- l'exécution d'une routine de « callback » ;
- l'occurrence d'une alarme si un message n'est pas fini d'envoyer ou reçu dans un délai de garde.

La transmission d'un message peut se faire de trois manières :

- sur demande de l'application lors de l'invocation du service d'envoi (SendMessage) ;
- périodiquement, dans ce cas une écriture sur le message en fait la mise à jour mais pas la transmission ; cette dernière est gérée entièrement par le sous système de communication ;
- en mode mixte : transmission périodique et envoi sur détection de changements significatifs pour la variable (la nature du changement est définie à la génération du système à l'aide de constructeurs de type >, <, = sur la valeur de la variable).

La surveillance temporelle des communications se fait à l'aide de chiens de garde associés à des alarmes.

Ainsi l'expiration d'un délai de garde, qui déclenche l'alarme peut être relayée par l'activation d'une tâche ou la signalisation d'une occurrence d'événement pour une tâche applicative. On peut signaler également la possibilité de mise en place de filtres prédéfinis configurables, aussi bien en réception qu'en transmission.

3.7. Prise en compte des interruptions

En réponse à une sollicitation externe se traduisant par une requête d'interruption, il y a exécution d'une routine de traitement (ISR : Interrupt Service Routine) en général dans le contexte de la tâche interrompue. Le but de cette routine est de prendre en compte la requête en l'acquittant sur le contrôleur correspondant puis de transformer cette requête en une action utilisable par les tâches de l'application, par exemple en signalant une occurrence d'un événement ou en activant une tâche. OSEK/VDX OS permet d'écrire des routines de service selon deux modèles :

- le premier (ISR type 1) est celui dans lequel on n'a pas besoin d'appeler un service exécutif. L'exécution du code de l'ISR est donc transparente pour l'exécutif et se traduit par un retard temporel de la tâche interrompue, durant le service de l'interruption ;
- le deuxième modèle (ISR type 2) est celui dans lequel la routine est déclarée comme une routine d'interruption par un mot clé spécifique (le générateur d'application génère alors les appels de service nécessaires pour signaler à l'exécutif l'entrée et la sortie d'une routine d'interruption). La majorité des services de l'API OS peuvent être utilisés dans le code de la routine sauf bien sûr les services bloquants.

Les traitements d'interruptions peuvent être imbriqués selon la priorité (matérielle) qui leur est associée et une commutation de contexte, par ré-ordonnancement, ne peut avoir lieu qu'à la fin de l'ISR (l'ISR ne peut être préemptée par une tâche). Il est possible à l'aide de deux services (SuspendOSInterrupts et ResumeOSInterrupts) de bloquer les interruptions des deux dernières catégories afin de protéger de courtes sections critiques.

Lors de l'appel d'un service à l'intérieur d'une ISR il n'y a pas de ré-ordonnancement pour laisser se terminer l'ISR. Celui-ci ne peut avoir lieu que lors de l'invocation du service de terminaison de l'ISR la plus « externe » dans l'éventuelle imbrication des ISRs. L'ordonnancement des interruptions est purement matériel, c'est le contrôleur qui le fixe. De ce fait, il ne faut pas laisser une ISR de type 1 terminer une séquence imbriquée d'interruptions car il n'y aura pas de ré-ordonnancement si des ISRs de type 2 ont invoqué des services exécutifs. Ce problème ne peut se résoudre que par le choix des priorités matérielles des interruptions. Ainsi, les ISRs de type 1 doivent être, matériellement, de priorité supérieure à celles de type 2.

3.8. Les classes de conformité

La notion de « Classe de conformité » permet d'adapter l'exécutif aux besoins différents des organes de contrôle et aux caractéristiques de supports d'exécution, depuis les petits microcontrôleurs avec peu de mémoire jusqu'à des unités de contrôle complexes et puissantes. Dans OSEK/VDX quatre classes de conformité ont été définies qui s'appuient sur les 3 attributs suivants : le type de tâche (basique ou étendue), la possibilité de mémoriser les requêtes d'activation pour une tâche et enfin le nombre de tâches par niveau de priorité. Les caractéristiques des quatre classes sont définies dans le tableau 1 :

- pour la classe BCC1 : des tâches basiques seulement, une requête unique d'activation de tâche, une seule tâche par niveau de priorité ;
- pour la classe BCC2 : des tâches basiques seulement, mémorisation de requêtes multiples d'activation de tâches, plusieurs tâches par niveau de priorité ;
- pour la classe ECC1 : des tâches basiques et des tâches étendues, une requête unique d'activation de tâche, une seule tâche par niveau de priorité ;
- pour la classe ECC2 : des tâches basiques et des tâches étendues, mémorisation de requêtes multiples d'activation pour les tâches basiques seulement, plusieurs tâches par niveau de priorité.

Selon la classe utilisée le modèle d'application correspondant varie de manière importante. Ainsi avec la classe BCC1 et des tâches périodiques indépendantes, on peut avoir un modèle analysable analytiquement. A l'opposé, on obtient un modèle complexe en classe ECC2 qui ne peut être analysé que par simulation.

4. Une approche différente : OSEKtime OS

Avec l'arrivée dans quelques années, dans le secteur automobile, des technologies « X by Wire » le besoin de systèmes plus déterministes devient crucial aussi bien pour les applications que pour les réseaux de communications. Ainsi, dans ce dernier cas la compétition fait actuellement rage entre des offres de réseaux sécuritaires tels que TTP/C [14], TTCAN [20] ou encore Flexay [21]. Le consortium OSEK a de son

côté fait la proposition OSEKtime [19] rapidement évoquée ici.

Une application OSEKtime peut comporter deux types de tâches :

- celles dites TT (Time Triggered) dont l'exécution est contrôlée par le noyau OSEKtime
- les autres qui ont les caractéristiques des tâches OSEK-VDX OS et peuvent donc invoquer les services précédents

Les tâches TT sont des modules (un point d'activation et un point de sortie). Elles ne possèdent que 3 états : « suspended » (elle est inactive) ; « running » (elle a le processeur) ; « preempted » (elle a été préemptée par une autre tâche TT).

Les tâches TT sont activées par le noyau à partir d'une table statique d'ordonnancement déterminée hors ligne. Cette table définit les débuts temporels d'exécution des tâches TT à partir d'un temps local, éventuellement synchronisé avec un temps global (système). Dès qu'une tâche doit être active elle le devient et préempte éventuellement une autre tâche TT qui n'aurait pas fini son exécution. Celle-ci la reprendra dès que le processeur redeviendra libre. L'ensemble des tâches TT doit s'exécuter dans le temps qui a été prévu hors ligne (le dispatcher round). Les échéances sont surveillées par le noyau au moyen de marques spécifiques dans la table d'ordonnancement. Une violation d'échéance entraîne l'exécution d'une routine utilisateur puis l'arrêt du système (ShutdownOS), ce qui paraît un peu étonnant !

Dans le « temps libre », non occupé par les tâches TT, les tâches utilisant le modèle d'application et les services présentés précédemment peuvent s'exécuter, mais, même si elles sont en mode non-préemptif elles seront systématiquement préemptées par les tâches TT.

5. Conclusion

Nous avons présenté les concepts de base d'un exécutif temps réel sans toutefois, compte tenu de la place disponible, prétendre avoir fait une analyse exhaustive : d'autres points seraient encore à aborder tels que la gestion de la mémoire, la gestion des exceptions et erreurs, un approfondissement de la communication inter-systèmes, ainsi que bien entendu l'approfondissement des exécutifs Unix temps réel et ceux académiques. Le noyau d'OSEK/VDX, un exemple d'exécutif statique, développé dans le contexte de « l'électronique embarquée » pour l'automobile, a été utilisé afin d'illustrer la nature des services que peut apporter un exécutif et donc montrer tout l'intérêt de la conception d'une application s'exécutant sous le contrôle d'un d'exécutif. C'est là une démarche naturelle puisque la programmation d'une application sous forme de tâches reflète bien le découpage fonctionnel que fait le concepteur lors de l'analyse du cahier des charges. Les produits commerciaux disponibles sont nombreux et, si initialement leur emploi était restreint aux produits de haute technologie, actuellement on peut constater une croissance très importante des domaines d'application.

6. Références

[1] Berry G., Couronne T., Gonthier G., « Programmation synchrone des systèmes réactifs : le langage Estere », *TSI*, vol. 6, n°4, p. 305-316, 1987.

[2] Harel D., Statecharts : « A Visual Formalism for Complex Systems », vol. 8, p. 231-274, North Holland, 1987.

[3] André C. : « L'approche synchrone pour le développement des systèmes temps réel », Chapitre 4 de la section Systèmes Temps Réel, Encyclopédie de l'informatique, Vuibert, à paraître en 2005.

[4] Kopetz H., Damm A., Koza C., Mulazzani M., Senft C., Zainlinger R., « The MARS approach », IEEE Micro, vol 9, n° 1, p. 25-40, 1989.

[5] Kligerman E., Stotenko A-D., « Real-time Euclid : a language for reliable real-time systems », IEEE TSE, vol.12, n°9, p. 941-949, 1986.

[6] Roux O., Creusot D., Cassez F., Elloy J-P., « Le langage réactif asynchrone Electre ». TSI, vol. 11, n°5, p. 35-66, 1992.

[7] Trinquet Y., Elloy J-P., « Systèmes d'exploitation temps réel : les principe »s, Techniques de l'Ingénieur, Traité « Contrôle et Mesures », R8 050, 1999.

[8] Trinquet Y., Elloy J-P., « Systèmes d'exploitation temps réel : exemples d'exécutifs industriels », Techniques de l'Ingénieur, Traité « Contrôle et Mesures », R8 052, 1999.

[9] Stankovic J.A. and Rajkumar R., " Real-Time Operating Systems ", The Journal of Real-Time Systems, Volume 28, Number 2/3, November/December, p. 237-253, 2004.

[10] Gallmeister B., « Programming for the real world : Posix 4 », O'Reilly & Associates Inc., 1995.

[11] RT-Linux, site web : http://www.rtlinux-gpl.org

[12] RTAI, site web : http://www.rtai.org

[13] Reisinger J., Steininger A. and Leber G., « Predictably Dependable Computing Systems », chapter The PDCS Implementation of MARS Hardware and Software, Springer-Verlag, pp. 209-224, 1995.

[14] TTTech , site web : http://www.tttech.com

[15] Stankovic J.A. and Ramamritham K., « The Spring Kernel : A New Paradigm for Real-Time Operating Systems », Operating Systems review, ACM Press, vol. 23, n°3, pp. 29-53, pp. 54-71, july 1989.

[16] Tokuda H. and Mercer C.W., « ARTS : A Distributed Kernel », Operating Systems review, ACM Press, vol. 23, n°3, pp. 29-53, july 1989.
[17] Sha L., Rajkumar R., Lehocsky.J-P., « Priority Inheritance Protocols ; an Approach to Real-Time Synchronisation », IEEE Trans. on Computers, vol. 39, n°9, p. 1175-1185, 1990.
[18] Jensen E.D., Locke C.D. and Tokuda H., « A Time-Driven Scheduling Model for Real-Time Operating Systems », Proc. of IEEE Real-Time Systems Symposium, pp. 112-122, 1985.
[19] OSEK Group., « OSEK/VDX Operating System Specification », site web : http://www.osek-vdx.org
[20] TT-CAN Bosch, site web : http://www.can.bosch.com
[21] Consortium Flexray , site web : http://www.flexray-group.org

Figures et tables

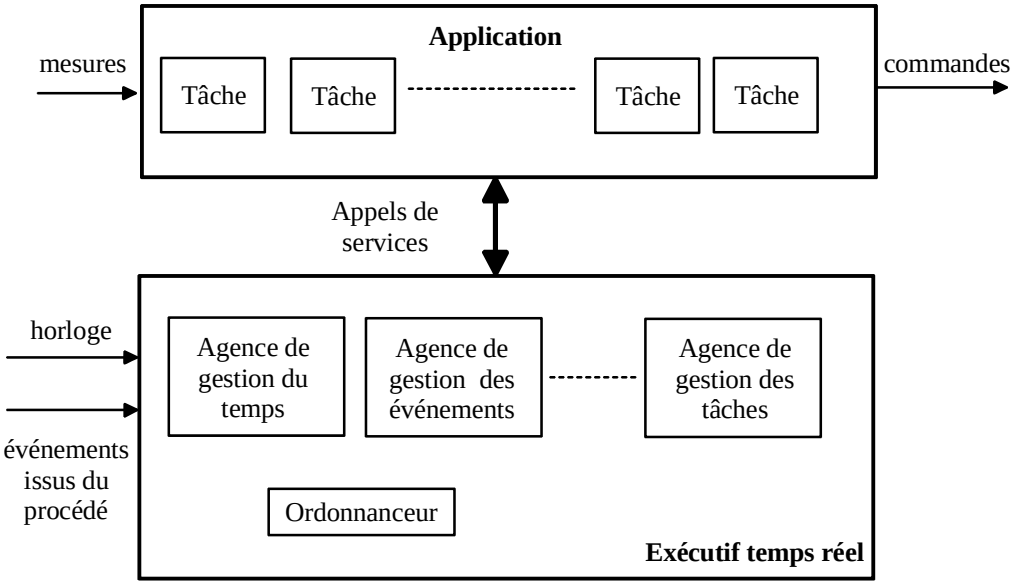


Figure 1. *Exécutif, application et environnement.*

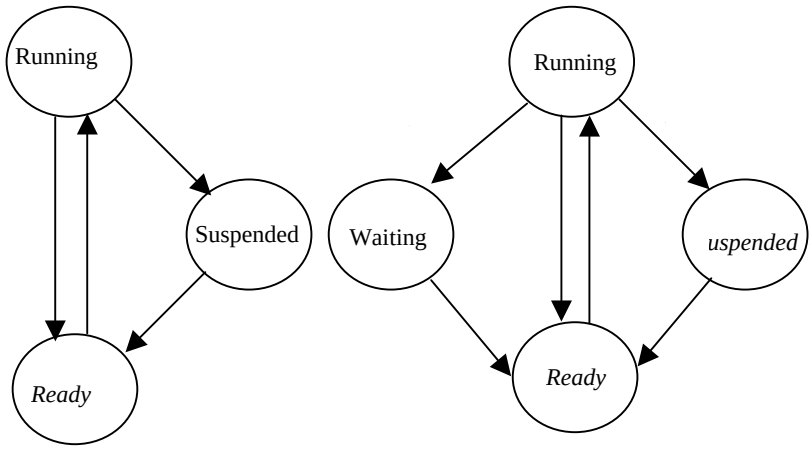


Figure 2. *Les états d'une tâche OSEK/VDX : basique (à gauche), étendue (à droite)*

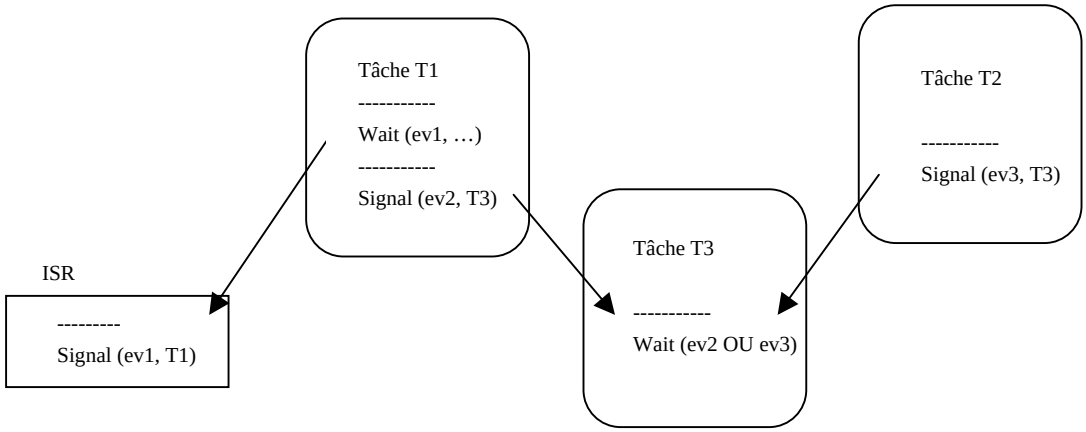


Figure 3. *Exemple de synchronisation par événement.*

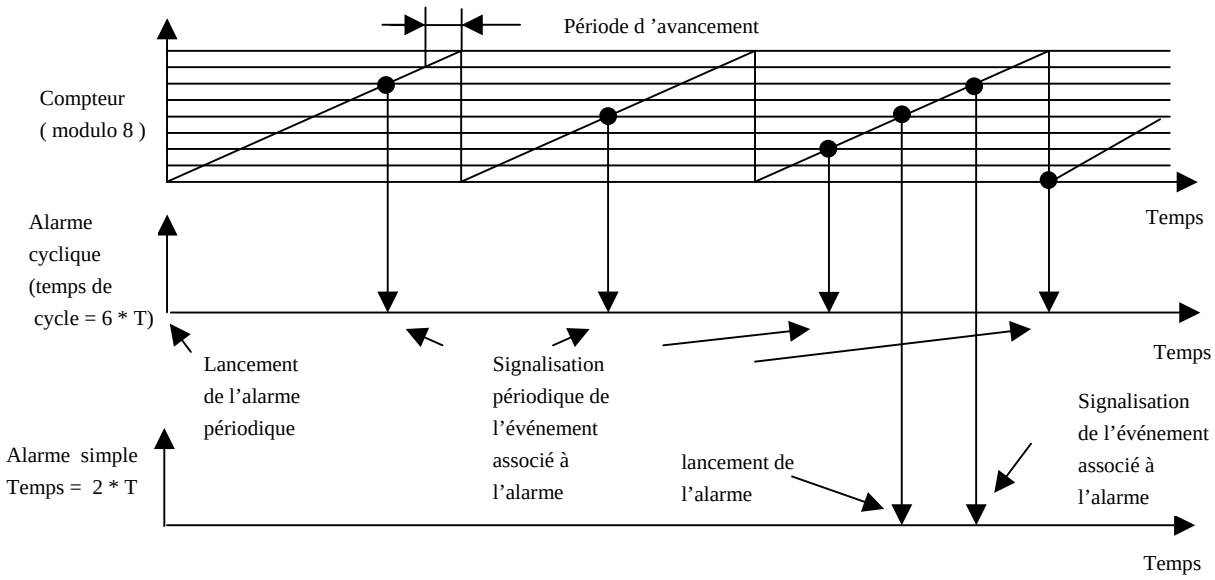


Figure 4. *Fonctionnement des alarmes.*

	Tâches basiques		**Tâches étendues**	
Classe	*Nb. de tâches par niveau*	*Nombre d'activations*	*Nb. de tâches par niveau*	*Nombre d'activations*
BCC1	1	1	----	----
BCC2	≥1	≥1	----	----
ECC1	1	1	1	1
ECC2	≥1	≥1	≥1	1

Tableau 1. *Les classes de conformité.*

Conditions de faisabilité pour l'ordonnancement temps réel préemptif et non préemptif

Laurent George
Ecole Centrale d'Electronique
53 rue de Grenelle
75007 Paris
lgeorge@ece.fr

Abstract

Cet article retrace les différents résultats disponibles dans l'état de l'art pour le dimensionnement d'un système temps réel monoprocesseur. Ces résultats sont basés sur l'expression de Conditions de Faisabilité qui permettent de garantir le respect de contraintes temporelles associées aux exécutions des tâches dans un système. Nous nous intéressons tout d'abord à la caractérisation d'un problème d'ordonnancement conduisant à la définition des modèles de tâches, de contraintes temporelles et d'ordonnancement d'un système temps réel. Nous nous intéressons ensuite aux principaux algorithmes d'ordonnancement issus de l'état de l'art en précisant les contextes pour lesquels ils sont optimaux. Nous étudions ensuite les scenarii pires cas ainsi que les conditions de faisabilité pour le respect de contraintes temporelles en contexte préemptif, non-préemptif pour les politiques d'ordonnancement basées sur des priorités fixes (Fixed Priority : FP), dynamiques (Dynamic Priority : DP) et hybride FP/DP de tâches ordonnancées selon FP avec un ordonnancement DP. Nous étudions en particulier FP/FIFO.

1. Introduction

La théorie de l'ordonnancement a été largement étudiée ces trente dernières années. De nombreux résultats sont disponibles. Ils permettent de résoudre le problème de dimensionnement d'un Système Temps réel. Le dimensionnement permet de vérifier avant sa conception qu'un système temps réel respecte des propriétés de ponctualité associées à l'exécution des tâches. Ce dimensionnement se base sur une approche "pire cas" qui permet de garantir que tout au long de la vie du système, les tâches respecteront leurs contraintes temporelles. L'approche pire cas se distingue des approches basées sur une analyse en moyenne ou par simulation. Vérifier qu'un STR est conforme à ses spécifications en pire cas permet de garantir la propriété de déterminisme : pour tout scénario d'activation possible, Les tâches respectent leurs contraintes temporelles.

La méthodologie généralement utilisée pour résoudre un problème d'ordonnancement est la suivante :

- Identifier la classe du problème à résoudre.
- Identifier les scenarii d'activation possibles des tâches dans le STR
- Identifier dans l'ensemble des scenarii d'activation possibles le sous ensemble conduisant aux pires temps de réponse des tâches (scenarii pires cas).
- Déterminer pour ce sous ensemble les Conditions de Faisabilité (CF) associées.
- Vérifier que les CF sont satisfaites pour une architecture donnée.

Cette méthodologie est plus formellement reprise dans [19]. La classe d'un problème temps réel est définie par le modèle d'ordonnancement utilisé, le modèle de tâches et le modèle de contraintes temporelles considéré. A partir de l'identification de la classe du problème à résoudre, nous pouvons identifier la complexité du problème à résoudre et conclure sur l'existence de solutions pour le problème.

Les scenarii d'activation dépendent des hypotheses faites sur les instants d'activation des tâches. On distingue deux approches : l'approche synchrone et l'approche asynchrone.

- L'approche synchrone impose les scenarii d'activation des tâches ou un motif de base, reproduit à l'identique pendant toute la durée de vie du système.
- L'approche asynchrone ne fait pas d'hypothèse particulière sur ces instants d'activation.

Le choix de l'approche synchrone peut se justifier dans le but de réduire la combinatoire des scenarii possibles ou pour éviter d'avoir à considerer des scenarii trop durs pour le respect des contraintes temporelles des tâches. Elle est intéressante si l'on peut démontrer que l'on ne retombe pas sur les scenarii pires cas que l'on souhaitait éviter. Ce qui conduit dans le cas général à résoudre le problème de savoir si il existe un instant où des tâches sont activées en même temps (problème de K-congruence simultanée [21]). Ce cas étant à la base des scenarii pires cas à considérer dans l'approche asynchrone (cf. section 4). L'approche asynchrone fournit des résultats plus généraux et plus robustes car elle permet de garantir le respect des contraintes temporelles en ne faisant pas d'hypothèse particulière sur les scenarii d'activation. Elle peut amener à valider le système sur un sous ensemble de scenarii pires cas pouvant avoir une probabilité d'occurrence faible mais offre une certaine souplesse pour le dimensionnement. Nous nous focalisons dans cet article sur cette approche.

Cet article est organisé comme suit. Dans la section 2 nous présentons les différents modèles et principes permettant d'identifier la classe du problème d'ordonnancement à résoudre. Nous décrivons les modèles de tâches, les modèles de contraintes temporelles, les modèles d'ordonnancement. Nous précisons les hypothèses et notations utilisées. La section 3 présente les différents algorithmes issus de l'état de l'art et précise les conditions selon lesquelles ils sont optimaux. La section 4 introduit la notion de période active et définit les scenarii qui conduisent au pire temps de réponse d'une tâche pour les politiques FP,DP et FP/DP. Ces scénarii pires cas sont à la base des tests de faisabilité présentés dans la section 5. La section 6 se focalise sur la politique d'ordonnancement FP/FIFO en contexte préemptif et non préemptif. Finalement nous concluons.

2. Caractérisation d'un problème d'ordonnancement

Non rappelons dans cette section quelques concepts classiques pour la caractérisation de la classe du problème d'ordonnancement.

2.1. Modèle de tâche

Nous considérons un ensemble $\tau = \{\tau_1, ..., \tau_n\}$ de n tâches. Différentes **lois d'arrivées** ont été étudiées dans l'état de l'art. On distingue les lois d'arrivées suivantes :

- l'arrivée *périodique* : les demandes d'activation de la tâche sont périodiques, de période T_i
- l'arrivée *sporadique* : les demandes d'activation de la tâche ont une inter-arrivée minimale $\geq T_i$
- l'arrivée *apériodique* : 1 seule demande d'activation de la tâche pendant la durée du système. Ce modèle a été largement étudié en présence de tâches périodiques temps réel. L'objectif est d'admettre des tâches apériodiques aux contraintes temps réel relatives (souples) en présence de tâche périodiques temps réel [34], [37].
- l'arrivée sur une *fenêtre glissante* : au plus n_i arrivées sur une fenêtre glissante W_i. Ce modèle s'étudie à l'aide du modèle sporadique : n_i arrivées sur une fenêtre W_i sont équivalentes à n_i tâches sporadiques indépendantes de période W_i. Cette loi d'arrivée est bien adaptée à la prise en compte des interruptions.

Nous nous intéressons dans cet article aux lois d'arrivées périodiques et sporadiques.

Le **modèle de stucture** d'une tâche précise les contraintes associées à la structure des tâches. On distingue :
La structure locale :

- La *séquence* : une tâche élémentaire, constituée d'un code séquentiel.
- La tâche avec *contraintes de précédences*, constituée d'un ensemble de tâches séquentielles avec relation de précédence linéaire ou en graphe.
- La tâche avec *contraintes de ressources*, pour laquelle une partie du code accède à une ressource critique en exclusion mutuelle. La section critique est non préemptive.

La structure distribuée, constituée d'un ensemble de tâches qui s'exécutent sur plusieurs nœuds interconnectés par un réseau de communication :

- *l'arbre*, pour lequel une tâche exécutée sur un nœud déclenche un ensemble de traitements sur d'autres nœuds, formant un arbre (exemple : le multicast) sans réponse de ces derniers
- *client/serveur* (ou requête/réponse)
- le *graphe* défini par un ensemble de relations de dépendances entre tâches distribuées.

Nous nous intéressons dans cet article au modèle de tâche en séquence sans contraintes de précédences ni de ressources.

Nous introduisons maintenant les différentes hypothèses envisageables sur les **instants d'activa-**

tion des tâches.

- Tâches **concrètes** : on impose un scénario d'activation particulier des tâches.
- Tâches **non concrètes** : on ne connaît pas à priori les instants d'activation des tâches.

Remarque: Pour la loi d'arrivée périodique, considérer un modèle de tâches concrètes revient à préciser les instants de première activation des tâches.

Nous introduisons maintenant les notations précisant la durée d'exécution d'une tâche (de type séquence) et le pire temps de réponse d'une tâche.

- La **durée pire cas d'exécution** d'une tâche τ_i (Worst Case Execution Time WCET) est dénotée C_i.
  - Cette durée d'exécution pire cas correspond à l'exécution de la tâche seule sans aucune préemption du système d'exploitation.
  - Plusieurs approches sont possibles pour déterminer le WCET d'une tâche [33]. Soit par analyse précise du code machine de la tâche et de l'architecture matérielle sur laquelle s'exécute la tâche. Soit par benchmark sur une architecture réelle, le C_i est plus difficile à garantir, elle dépend des conditions d'exécution (type de mémoire, cache, ...).
- le **pire temps de réponse** r_i d'une tâche τ_i est la pire durée entre la demande d'activation de τ_i et sa terminaison effective. r_i est fonction de la politique d'ordonnancement et tient compte du retard introduit par les tâches plus prioritaires que τ_i (cf. section 5). Nous avons toujours $r_i \geq C_i$.

Nous introduisons maintenant la notion de **gigue** d'une tâche τ_i dénotée J_i

- La gigue est un délai entre un événement et la prise en compte de cet événement par le système.
- On parle de gigue d'activation lorsque cet événement correspond à une demande d'activation d'une tâche (cf. figure 1).
- Dans un contexte réseau, la gigue permet de modéliser pour un flux les variations sur les temps de traitements des nœuds visités par ce flux et sur les délais des liens de communication. Elle est utilisée pour le calcul du temps de réponse de bout en bout d'un flux ([40] pour l'approche holistique, [26] pour l'approche par trajectoire, [18] pour le network calculus).

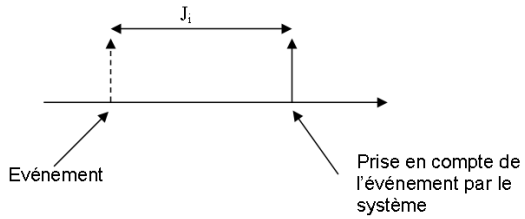


FIG. **1.** *Gigue d'activation d'une tâche*

Dans le but de simplifier les notations, nous ne considérons pas dans cet article la gigue d'activation. Les scénarii pires cas de la section 4.2 et les tests de faisabilité énoncés dans la section 5 peuvent néanmoins facilement être adaptés pour prendre en compte ce paramètre (cf. par exemple [38], [10]).

2.2. Modèles de contraintes temporelles

L'objectif de ces approches est de garantir le respect de contraintes temporelles pour l'exécution des tâches du système. On distingue différents modèles de contraintes temporelles :

L'**échéance relative** d'une tâche τ_i dénotée D_i : contrainte temporelle associée à la tâche τ_i.
L'**échéance absolue de terminaison au plus tard** :

- Toute tâche τ_i activée à l'instant t doit impérativement être terminée à l'instant $t + D_i$

L'**échéance absolue de démarrage au plus tard** :

- Toute tâche τ_i activée à l'instant t doit impérativement avoir débutée son exécution à l'instant $t + D_i$

La **fonction de valeur** (Time Value Function) pour modéliser le gain associé à l'exécution d'une tâche temps réel ([30], [1], [5]).
Nous nous intéressons dans la suite au modèle de contrainte d'échéance absolue au plus tard.

2.3. Modèle d'ordonnancement

Nous introduisons maintenant les différentes contraintes envisageables pour l'ordonnancement des tâches.

Ordonnancement préemptif ou non-préemptif d'une tâche.

- Ordonnancement **préemptif** : L'ordonnanceur peut interrompre une tâche en cours d'exécution au profit d'une autre tâche.
- Ordonnancement **non préemptif** : L'ordonnanceur doit attendre la fin d'exécution de la tâche pour réordonnancer les tâches.

On parle **d'effet non-préemptif** le fait qu'une tâche moins prioritaire retarde une tâche plus prioritaire (cf. figure 2). Cet effet se produit lorsque qu'une tâche prioritaire est activée alors qu'une tâche non préemptive moins prioritaire est en cours d'exécution. La figure 2 présente la cas d'une tâche plus prioritaire que la tâche en cours mais exécutée après celle-ci.

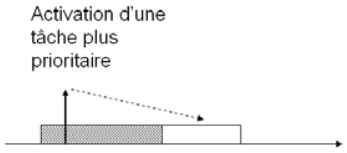


FIG. **2.** *Effet non préemptif*

Nous considérons dans cet article les deux types d'ordonnancement.

Nous précisons maintenant les différents modèles d'invocation de l'ordonnanceur :

- Event Driven : L'ordonnanceur est invoqué sur réception d'un évènement (nouvelle activation d'une tâche, traitement d'une IT, ...).
- Time Driven : L'ordonnanceur détermine ses instants d'invocation indépendamment des événements qui se produisent dans le système.

Avec l'ordonnancement Event driven, un coût lié au changement de contexte ou à la préemption de l'ordonnanceur est à envisager. Nous considérons dans cet article que ce coût est inclus dans la durée d'exécution pire cas de la tâche.

L'ordonnancement Time Driven est généralement réalisé à l'aide d'un ordonnanceur périodique. La période courante est de $10ms$ pour les systèmes d'exploitation non temps réel et de $1ms$ pour les systèmes d'exploitation temps réel. La valeur de la période a plusieurs impacts :

- Un retard sur la prise en compte d'une tâche qui se traduit par une gigue d'activation (cf. 2.1).
- Un coût supplémentaire (overhead) lié à l'exécution de l'ordonnanceur. L'overhead limite la valeur de la période de l'ordonnanceur ([6]).

Nous considérons dans la suite un ordonnancement event driven.

L'ordonnanceur doit déterminer l'ordre dans lequel les tâches sont exécutées. Une solution très classique est d'affecter des priorités aux tâches, ces priorités sont utilisées pour définir cet ordre. L'ordonnanceur lorsqu'il est invoqué choisit la tâche de plus haute priorité (Highest Priority First : HPF). On distingue la priorité fixe ou dynamique :

- **Priorité Fixe** : la priorité d'une tâche est identique pour toutes ses activations.
- On notera dans la suite **FP** (Fixed Priority) les algorithmes d'ordonnancement basés sur des priorités fixes avec l'algorithme en ligne HPF. Nous présentons les algorithmes FP les plus plus classiques dans le section 3.1.
- **Priorité dynamique** : la priorité d'une tâche est déterminée pour chaque activation de la tâche.
- On notera **DP** les algorithmes d'ordonnancement à base de priorités dynamiques. Nous focalisons dans cet article sur les conditions de faisabilité pour les ordonnancements FIFO (la priorité d'une tâche est en fait son instant d'activation même si FIFO n'a pas besoin explicitement de cette information) et Earliest Deadline First (EDF) dans la section 3.2.
- **FP/DP** est une version hydride d'ordonnancement mixte FP comme premier critère d'ordonnancement dénotant l'importance d'une tâche et DP comme second critère pour les tâches de même priorité fixe (cf. 3.3).

Propriété 1 *Dans cet article nous supposons pour* FP *et* DP *que la priorité la plus élevée correspond à la valeur de priorité la plus faible.*

Nous étudions à titre d'exemple dans le chapitre **??** la politique FP/FIFO et montrons que la prise en compte de la politique d'ordonnancement locale permet d'améliorer les conditions de faisabilité FP.

Nous introduisons maintenant la notion d'oisiveté d'un ordonnanceur.

- Un ordonnanceur est dit **oisif** si il ne traite pas nécessairement les tâches dès qu'elles sont activées alors qu'il n'a rien à faire.
  - Dans le contexte réseau, le but recherché en général est de réguler (Shaping) les instants d'entrée ou de sortie des tâches d'un noeud de communication.
  - On distingue deux types de régulateurs : le régulateur de débit (pour éviter les surcharges en entrée ou en sortie du noeud) et le régulateur de gigue (pour réduire la gigue des tâches en entrée d'un nœud dans un réseau).
- Un ordonnanceur **non oisif** est donc un ordonnanceur qui ne peut retarder l'exécution d'une tâche s'il n'a rien à faire. Ce type d'ordonnanceur correspond au cas de la plupart des systèmes d'exploitation temps réel.

Nous introduisons maintenant la notion d'optimalité d'un algorithme d'ordonnancement :

- Un algorithme est dit **optimal** par rapport à une classe de problème d'ordonnancement si il trouve une solution au problème lorsque qu'il en existe une.

Contraposée de cette définition :

- Si un algorithme d'ordonnancement optimal pour une classe de problème donnée ne trouve pas de solution alors il n'existe pas de solution au problème d'ordonnancement.

2.4. Concepts et notations

L'analyse menée considère le temps comme **discret** : les instants de demandes d'activation des tâches ainsi que les paramètres des tâches sont exprimés à l'aide d'une même unité : le tick d'horloge. [3] démontrent la pertinence de cette hypothèse par les faits suivants :

- la plupart des ordonnanceurs utilisent une horloge comme référence temporelle ;
- si tous les paramètres temporels des tâches sont exprimés à l'aide du même tick d'horloge, alors, ils démontrent qu'il existe une solution à base d'ordonnancement continu si et seulement si il en existe une en temps discret (dont le pas est le top d'horloge).

Ainsi, lorsque les paramètres des tâches sont entiers (multiples du même top d'horloge), il n'est pas restrictif de considérer que les instants d'ordonnancement le sont également. Nous considérons dans la suite le temps discret.

Soit $\tau = \tau_1 \dots \tau_n$ un jeu de n tâches sporadiques. Nous rappelons quelques définitions associées à ce jeu de tâches lorsque les tâches sont sans gigue d'activation. De plus $\forall x \in \mathcal{R}$, $\lfloor x \rfloor$ désigne la partie entière de x et $\lceil x \rceil$ désigne la partie entière supérieure de x.

- Un jeu de tâches périodique est dit **synchrone** si l'instant de demande de première activation coincide pour toutes les tâches autrement il est dit asynchrone.
- $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ est le **facteur d'utilisation du processeur**, il représente le pourcentage d′utilisation du processeur pour l'exécution d'un jeu de tâches [23]. Remarque : Une condition nécessaire évidente est : $U \leq 1$
- La **demande processeur** ([4, 35]) $h(t)$ est la somme des durées d'exécution requises par toutes les tâches dans le scénario synchrone dont l'instant d'activation et l'échéance absolue sont dans l'intervalle $[0, t]$:
  $h(t) = \sum_{D_i \leq t} (1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor) C_i$

La demande processeur est une mesure de la quantité de travail minimum que l'on doit exécuter pour toujours respecter les échéances des tâches.

- La **charge de travail** demandée par une tâche τ_i synchrone dans l'intervalle $[0, t[$, si τ_i est synchrone, est : $\left\lceil \frac{t}{T_i} \right\rceil C_i$.
- Par analogie, la charge de travail demandée par une tâche τ_i synchrone dans l'intervalle $[0, t]$, est : $(1 + \left\lfloor \frac{t}{T_i} \right\rfloor) C_i$.
- Soit τ' un ensemble de tâches et $x \in \mathcal{R}$. Par convention, $max^*_{\tau'}(x)$ désigne la valeur maximale du paramètre x dans τ' si $\tau' \neq \emptyset$ et 0 sinon.

3. Algorithmes d'ordonnancement / optimalité

3.1 Algorithmes à priorités fixes

Ordonnancement Rate Monotonic (RM) [23] :

- La priorité est fonction de la période(T_i). Plus la période est petite, plus la tâche est prioritaire

Optimalité :

- Elle n'est pas générale (on peut trouver des solutions faisables avec un ordonnancement DP mais non faisable avec RM).
- RM est optimal pour l'ordonnancement des tâches sporadiques ou périodiques préemptives si $\forall \tau_i, i = 1 \dots n, T_i = D_i$
- RM n'est plus optimal en contexte préemptif si $\forall i = 1 \cdots n, D_i \leq T_i$ ou dans le cas général où les périodes et les échéances sont indépendantes
- RM n'est pas optimal pour l'ordonnancement de tâches non-préemptives [10].

Ordonnancement Deadline Monotonic (DM) : [21]

- La priorité d'une tâche est fonction de son échéance relative(D_i).
- Plus l'échéance relative est petite, plus la tâche est prioritaire.

Optimalité :

- Elle n'est pas générale (comme pour RM).
- DM est optimal pour l'ordonnancement préemptif de tâches sporadiques ou périodiques si pour toutes les tâches $\forall i = 1 \dots n, D_i \leq T_i$
- DM n'est pas optimal en contexte non-préemptif en général.
- DM est optimal en non-préemptif ([11, 10]) si $\forall i = 1 \dots n, D_i \leq T_i$ et $\forall (i, j), C_i \leq C_j \Rightarrow D_i \leq D_j$

FP (Fixed Priority with highest priority first)

- Cet algorithme est à utiliser lorsqu'il n'existe pas de relation évidente valable pour toutes

les tâches entre la période et l'échéance permettant de conclure sur l'optimalité d'une attribution particulière des priorités
- ou lorsque les priorités sont imposées.

On définit pour FP, pour une tâche τ_i, les ensembles suivants contenant respectivement les tâches plus prioritaires, moins prioritaires et de même priorité que τ_i.

On rappelle que la priorité la plus grande est celle de valeur la plus faible.

- P_i, la priorité d'une tâche τ_i
- $hp_i = \{\tau_j, j \in [1,n]$, tel que $P_j < P_i\}$: l'ensemble des tâches de priorité supérieure à τ_i
- $\overline{hp_i} = \{\tau_j, j \in [1,n]$ tel que $P_j > P_i\}$: l'ensemble des tâches moins prioritaires que τ_i
- $sp_i = \{\tau_j, j \in [1,n], j \neq i$, tel que $P_j = P_i\}$

Optimalité :

- Audsley dans [2] propose une méthode d'attribution optimale des priorités : valable pour avec FP pour l'ordonnancement préemptif de tâches sporadiques ou périodiques.
- Cette méthode est aussi valable en contexte non préemptif (cf. [10]).
- cette optimalité n'est pas générale, comme pour RM et DM.

Principes de l'algorithme :

- Cet algorithme d'attribution fonctionne de manière itérative de la priorité la plus basse ($prio = n$) à la priorité la plus haute ($prio = 1$).
- La fonction cherche-si-faisable($\tau, prio$) détermine si dans l'ensemble courant de tâches τ, il existe une tâche ordonnançable sur le niveau de priorité $prio$. Elle s'appuie sur le calcul du temps de réponse maximum d'une tâche (cf. section 5). Elle retourne la première tâche ordonnançable τ_j trouvée ou 0 si aucune tâche n'est trouvée, auquel cas, il n'existe pas de solution au problème d'ordonnancement.
- Il faut noter que si sur un niveau de priorité il existe plusieurs tâches ordonnançables, on peut choisir arbitrairement une des tâches faisables sans remettre en cause l'optimalité de l'attribution des priorités.

Nous décrivons l'algorithme proposé par [2]

```
τ = {τ1, ..., τn} : task set ;
prio ← n : entier ; j : entier
failed← false : booléen ;
Tant que (τ ≠ ∅) faire
    j=cherche-si-faisable(τ,prio) ;
    Si (j ≠ 0 ET failed=false) Alors
        [on attribue la priorité prio à la tâche τj]
        attribue-priorité(j,prio) ;
        [on retire τj de l'ensemble τ]
        τ = τ − {τj} ;
        prio ← prio-1 ;
    Sinon
        failed=true ;
    Fin Si
Fait
```

Algorithme 1: Attribution optimale des priorités

Concernant l'ordonnancement priorité fixe préemptif ou non préemptif, la non faisabilité d'un problème d'ordonnancement en contexte préemptif n'implique pas que le problème ne soit pas faisable en contexte non préemptif (cf. [11]) et réciproquement.

3.2 Algorithmes dynamiques DP

Dans l'état de l'art on retrouve principalement quatre algorithmes d'ordonnancement : Earliest Deadline First, First In First Out (FIFO), Least Laxity First (LLF) et Round Robin (RR) (ou ses variantes). Ces algorithmes utilisent une priorité explicite (EDF, LLF) ou implicite (FIFO, RR) pour l'ordonnancement des tâches. La priorité implicite n'est pas nécessairement utilisée par l'algorithme d'ordonnancement mais est utilisée pour établir les conditions de faisabilité associées.

- Pour des algorithmes à priorités dynamiques, on définit la priorité dynamique à un instant $t \geq t_i$ d'une tâche τ_i activée à l'instant t_i par $P_i(t, t_i)$.

Ordonnancement EDF : [23]
La priorité d'une tâche est son échéance absolue de terminaison au plus tard (cf. section 2.2). Si t_i est l'instant d'activation d'une tâche τ_i alors sa priorité est $\forall t \geq t_i, P_i(t, t_i) = t_i + D_i$. La tâche la plus prioritaire est celle d'échéance absolue la plus petite.

Optimalité :

- EDF est optimal pour l'ordonnancement de tâches sporadiques ou périodiques préemptives ([9]).

- EDF est optimal pour l'ordonnancement de tâches sporadiques ou périodiques non préemptives non concrètes ([14]).
- Cette optimalité n'est plus valable pour des tâches périodiques concrètes non préemptives ([14]).

Cet algorithme présente des résultats d'optimalité très intéressants. Il est cependant instable en cas de surcharge. Un effet avalanche peut se produire : en cas de surcharge, on passe son temps à executer des tâches qui ratent leurs échéances. Des solutions permettant de gérer les situations de surcharges sont proposées dans [24], [7], [17]. Le principe étant en cas de surcharge de basculer sur des algorithmes à priorités fixes et d'éliminer un sous ensemble de tâches pour stabiliser le système.

Ordonnancement FIFO :
FIFO peut être considéré comme un algorithme à priorité dynamique (la priorité d'une tâche est en fait son instant d'activation même si FIFO n'a pas besoin explicitement de cette information). La priorité implicite d'une tâche τ_i activée en t_i est $\forall t \geq t_i, P_i(t, t_i) = t_i$.

Optimalité :
- FIFO n'est pas optimal pour l'ordonnancement de tâches périodiques ou sporadiques
- FIFO est optimal pour l'ordonnancement de tâches non concrètes lorsque les tâches ont les mêmes contraintes temporelle (échéance). FIFO se comporte exactement comme EDF dans ce cas.
- FIFO est optimal pour minimiser le temps de réponse maximum d'un ensemble de tâches de même importance ([13]).

Ordonnancement LLF : ([29])
La tâche la plus prioritaire à un instant t est la tâche de plus faible laxité. Pour toute tâche τ_i activée à l'instant t_i, la laxité est définie à l'instant t par : $t_i + D_i - (t + C_i(t))$ où $C_i(t)$ est la durée d'exécution restante pour la tache τ_i à l'instant t (cf. figure 3). La priorité de τ_i est donc $\forall t \geq t_i, P_i(t, t_i) = t_i + D_i - (t + C_i(t))$
Cet algorithme présente l'inconvénient, lorsque deux tâches ont la même laxité, de conduire à un grand nombre de changements de contextes (on passe alternativement d'une tâche à l'autre).
Optimalité :
- LLF est optimal pour l'ordonnancement de tâches préemptives périodiques ou sporadiques ([29]).
- LLF n'est pas optimal pour l'ordonnancement de tâches non préemptives ([10]).

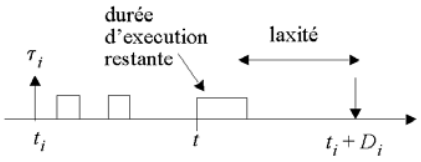


FIG. **3.** *Algorithme* LLF

Ordonnancement RR : [28]
Cet algorithme est un des algorithmes définis par la norme Posix 1003.1.b. (SCHED_RR s'applique aux tâches de même priorité). Principes :
- Chaque tâche se voit attribuer un quantum de temps pour son exécution.
- Après consommation de son quantum, la tâche est placée en fin de la file d'attente.
- L'ordre d'exécution est donc cyclique, les tâches sont exécutées à tour de rôle à concurrence de leur quantum.

Optimalité :
- Cet algorithme n'est pas optimal pour l'ordonnancement de tâches périodiques ou sporadiques.
- Mais il est possible de trouver des configurations ordonnançables avec RR alors qu'il n'existe pas de solution avec FP ([31]).

Dans un contexte réseau, des variantes de RR on été proposées basées sur le Weigthed Fair Queeing (WFQ) qui propose un ordonnancement RR pondéré par des poids fonction du débit souhaité pour le flux ([32] et [8]).

3.3 Algorithmes hybrides

Nous nous intéressons dans cette partie aux algorithmes FP/DP combinants un ordonnancement de type priorité fixe (FP) comme critère principal, où la priorité fixe dénote le degré d'importance ou la criticité de la tâche, et un ordonnancement dynamique (Dynamic Priority : DP) pour les tâches de même priorité fixe.

Nous introduisons la notion de **priorité généralisée** à un instant t, pour une tâche τ_i activée à un instant t_i :
- Cette priorité est égale à P_i comme premier critère d'ordonnancement si τ_i est en compétition avec des tâches de priorités fixes différentes ou
- $P_i(t, t_i)$ comme second critère d'ordonnancement, si τ_i entre en compétition avec des tâches de même priorité fixe.

A un instant t, une tâche τ_i activée en t_i est plus prioritaire qu'une tâche τ_j activée en t_j si :

$$\begin{cases} P_i < P_j \ \ ou \ si \\ \\ P_i = P_j \ \ et \ \ P_i(t,t_i) \leq P_j(t,t_j) \end{cases}$$

Exemple d'algorithmes FP/DP *:*

- FP/FIFO : Un implémentation classique de FP avec un ordonnancement FIFO sur un niveau de priorité fixe.
- FP/RR : RR pour les tâches de même priorité fixe.
- FP/LLF : Dénommé également MUF (Maximum Urgency First) développé par [36].
- FP/EDF : Où EDF est utilisé pour ordonnancer les tâches de même priorité fixe à l'aide d'un second critère d'échéance. Cet algorithme permet de décorréler l'importance d'une tâche de son échéance. [27] montrent que FP/EDF domine FP/FIFO sous certaines hypothèses.

Posix 1003.1.b. est une implémentation FP/DP de trois politiques d'ordonnancement DP : FIFO,RR et OTHER (ordonnancement configurable, généralement FIFO par défaut [31]).

[22] présentent un algorithme dans une contexte d'ordonnancement de messages dénommé RPQ (*Rotating Priority Queues*), pour lequel les tâches sont ordonnancées par priorités fixes FP et les tâches de même priorité sont ordonnancées FIFO. L'idée principale est d'approximer l'algorithme EDF en faisant évoluer périodiquement les priorités des files FIFO. Contrairement à EDF, cet algorithme ne nécessite pas un tri pour insérer une nouvelle tâche.
Nous nous intéressons dans le chapitre 6 à l' algorithme d'ordonnancement FP/FIFO en contexte préemptif et non préemptif.

4 Périodes actives et scenarii pires cas

4.1 Périodes actives

Nous introduisons tout d'abord la notion d'**instant oisif** et de **période active**, à la base de la plupart des conditions de faisabilité pour l'ordonnancement temps réel monoprocesseur.
Instant oisif :

- Un instant oisif (idle time) est un instant t tel qu'il n'y a plus de tâches activées avant t non terminée à l'instant t.

Période active :

- Un période active (busy period) est un intervalle de temps $[a, b[$ tel que a et b sont deux instants oisifs et qu'il n'y a pas d'instant oisif dans $]a, b[$.

[16] montrent que la première période active du scénario synchrone lorsque les tâches sont activées avec leur plus grande densité (périodiques) est la plus longue période active possible. Soit L la durée de cette période active.

L est solution de $L = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil C_i$.

Cette équation peut être résolue par la détermination du premier point fixe de la suite :

$$\begin{cases} L^{m+1} = \sum_{i=1}^{n} \left\lceil \frac{L^m}{T_i} \right\rceil C_i \\ L^0 = \sum_{i=1}^{n} C_i \end{cases}$$

Les notions d'instants oisifs et de périodes actives ont été étendues pour une politique d'ordonnancement donnée.

Dans le cas d'ordonnancement* FP *:

[20] introduit la notion de période active de niveau de priorité P_i pour une tâche τ_i.
Un instant oisif de niveau de priorité P_i en référence à une tâche τ_i ordonnancée ¨par FP est défini comme suit :

- Un instant oisif de niveau de priorité P_i est un instant t tel qu'il n'y a plus de tâches dans $\{hp_i \cup sp_i \cup \{\tau_i\}\}$ activées avant t et non terminées à l'instant t.

Une période active de niveau de priorité P_i est alors définie comme suit :

- Un période active de niveau de priorité P_i est un intervalle de temps $[a, b[$ tel que a et b sont deux instants oisifs de niveau de priorité P_i et qu'il n'y a pas d'instant oisif de niveau de priorité P_i dans $]a, b[$.

[10], [11] et [25] et montrent que la définition de la période active de niveau de priorité P_i peut être étendue en contexte non-préemptif on tenant compte de l'effet non préemptif maximum obtenu en activant juste avant le début de la période active de niveau P_i, une tâche τ_k de durée maximum et de priorité inférieure à cette de τ_i.

Dans le cas d'ordonnancement* DP *ou* FP/DP *:

La notion d'instant oisif a été généralisé ([27]) pour des ordonnanceurs de type DP et FP/DP qui respectent la propriété suivante :

Propriété 2 *La priorité d'une tâche τ_i activée en t_i est dite* ***invariante*** *si elle vérifie les propriétés suivantes (i) $\forall t \geq t_i$, la priorité de τ_i activée en t reste égale à celle de τ_i en t_i, et (ii) la priorité de τ_i ne peut pas décroître si t_i décroît.*

Les algorithmes EDF, FIFO, FP/FIFO et FP/EDF respectent cette propriété. Nous considérons dans la

suite pour les algorithmes DP et FP/DP les algorithmes d'ordonnancement qui respectent la propriété 2.

Définition 1 *La priorité généralisée $PG_i(t_i)$ d'une tâche τ_i activée en t_i est définie par :*

- *P_i si τ_i est en compétition avec une tâche de priorité fixe différente.*
- *$\forall t \geq 0, P_i(t_i, t_i)$ sinon.*

Remarque: Si l'on n'est dans le cas DP uniquement, alors la notation $PG_i(t_i)$ se simplifie en $PG_i(t_i) = P_i(t_i, t_i)$

Définition 2 *Un instant oisif t' de niveau de priorité $PG_i(t_i)$ est un instant tel que toutes les tâches ayant une priorité généralisée supérieure ou égale à $PG_i(t_i)$ et arrivées avant t' sont terminées en t'.*

Définition 3 *Une période active de niveau $PG_i(t_i)$ est un intervalle de temps $[t', t'')$ tel que t' et t'' sont deux instants oisifs de niveau $PG_i(t_i)$ et qu'il n'existe pas d'instant oisif de niveau de priorité $PG_i(t_i)$ dans l'intervalle $]t', t''[$.*

Les conditions de faisabilité sont établies en identifiant les pires périodes actives qui conduisent aux pires conditions de faisabilité d'une tâche.

4.2 Scenarii pires cas

Nous énonçons maintenant les principes d'établissement du scénario pire cas pour une tâche τ_i activée à l'instant t_i permettant de calculer le temps de réponse.

- En contexte préemptif on détermine l'instant de fin d'exécution de la tâche. Si W_{i,t_i} est l'instant de fin d'exécution de τ_i alors son temps de réponse est $W_i(t_i) - t_i$.
- en contexte non préemptif, une tâche qui a débuté son exécution ne peut plus être interrompue. On cherche donc à calculer son instant de démarrage. Si $\overline{W}_i(t_i)$ est son instant pire cas de démarrage alors le temps de réponse de τ_i activée en t_i est : $\overline{W}_{i,t_i} + C_i - t_i$

Nous donnons les valeurs de $W_i(t_i)$ et de $\overline{W}_i(t)$ dans la chapitre suivant pour les politiques FP, FIFO, EDF, FP/FIFO et FP/EDF en contexte préemptif et non préemptif.

Ces calcul de temps de réponse utilisent la propriété suivante :

Propriété 3 *Les conditions de faisabilité pires cas sont obtenues lorsque les tâches sont à leur densité maximale (périodiques).*

Pour l'ordonnancement FP :
-Scénario pire cas FP préemptif :

Propriété 4 *Le pire temps de réponse d'une tâche τ_i est obtenu pour l'ordonnancement FP en contexte préemptif dans la première période active de priorité P_i du scénario pire cas où toutes les tâches de priorité supérieure ou égale à τ_i sont synchrones au début de la période active (cf. figure 4)*

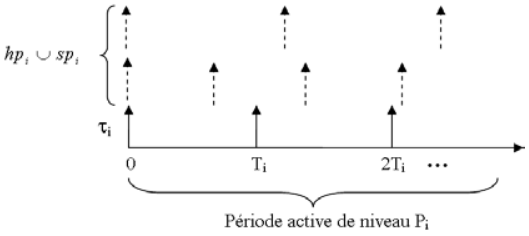


FIG. **4.** *Scénario pire cas préemptif FP pour une tâche τ_i*

[20] montre en contexte préemptif que la plus grande période active L_i de niveau de priorité P_i est obtenue pour une tâche τ_i dans le scénario 4. De plus, L_i est alors solution de :

$$L_i = \sum_{\tau_j \in hp_i \cup sp_i \cup \{\tau_i\}}^{n} \left\lceil \frac{L_i}{T_j} \right\rceil C_j.$$

-Scénario pire cas FP non préemptif :

Cette propriété est étendue en contexte non préemptif comme suit :

Propriété 5 *Le pire temps de réponse d'une tâche τ_i est obtenu pour FP en contexte non préemptif dans la première période active de priorité P_i du scénario où toutes les tâches de priorité supérieure ou égale à τ_i sont synchrones au début de la période active et une tâche moins prioritaire de durée maximum est activée un tick d'horloge (en -1) avant le début la période active de niveau P_i (cf. figure 5)*

L_i est alors solution de :

$$L_i = \sum_{\tau_j \in hp_i \cup sp_i \cup \{\tau_i\}}^{n} \left\lceil \frac{L_i}{T_i} \right\rceil C_i + max^*_{\tau_k \in \overline{hp_i}}(C_k - 1)$$

Remarque: Pour la tâche τ_i la moins prioritaire, en contexte préemptif ou non-préemptif, $L_i = L$. Pour trouver le pire temps de réponse de τ_i il faut donc tester dans son scénario pire cas les activations de τ_i en $0, T_i, 2T_i \ldots \left\lfloor \frac{L_i}{T_i} \right\rfloor$. Cette propriété est à la base des tests de faisabilité présentés dans la section 5.1.

Pour l'ordonnancement DP ou FP/DP :

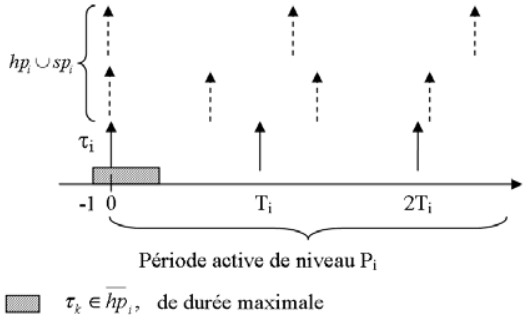


FIG. **5.** *Scénario pire cas non préemptif* FP *pour une tâche* τ_i

Nous nous intéressons dans cette section aux algorithmes DP et FP/DP qui vérifient la propriété 2. Cette propriété s'applique aux algorithmes EDF, FIFO, FP/FIFO et FP/EDF. Notons que cette propriété s'applique également à FP. Cependant, dans le cas de FP , les scenarii pires cas présentés précédemment sont plus simples.

Soit une période active de niveau $PG_i(t_i)$ dans laquelle une tâche τ_i est activée à l'instant t_i.

Propriété 6 *Une tâche* τ_j *est dite* ***potentiellement plus prioritaire*** *si il existe au moins un instant d'activation* t_j *de* τ_j *pour lequel* τ_j *soit plus prioritaire que* τ_i *activée en* t_i *dans cette période active.*

D'après la propriété 2, τ_j est déclarée potentiellement plus prioritaire que τ_i activée en t_i si la priorité de τ_j activée en $t_j = 0$, début de la période active de niveau $PG_i(t_i)$, est supérieure à celle de τ_i activée en t_i. On détermine formellement les ensembles :

- $hp_i(t_i) = \{\tau_j, j \neq i, \ tel \ que \ PG_j(0) \leq PG_i(t_i)\}$
- $\overline{hp_i}(t_i) = \{\tau_j, \ tel \ que \ PG_j(0) > PG_i(t_i)\}$

Les tâches $\tau_j \in hp_i(t_i)$ sont dans l'ensemble des potentiellement plus prioritaires que τ_i activée en t_i, les tâches $\tau_j \in \overline{hp_i}(t_i)$ sont moins prioritaires que τ_i.

Pour l'ordonnancement FP/DP, nous introduisons les notations complémentaires suivantes pour les tâches de même priorité fixe que la tâche τ_i (dans sp_i) ordonnancées DP.

- $sp_i(t_i) = \{\tau_j \in sp_i$, tel que $P_j(0,0) \leq P_i(t_i,t_i)\}$;
- $\overline{sp_i}(t_i) = \{\tau_j \in sp_i$, tel que $P_j(0,0) > P_i(t_i,t_i)\}$.

-Scénario pire cas DP ou FP/DP préemptif :

On notera que DP est un cas particulier de FP/DP si l'on n'a qu'une seule priorité fixe. Nous nous focalisons donc dans la suite sur FP/DP.

La propriété suivante ([10],[25]) détermine le scénario conduisant au pire temps de réponse pour une tâche τ_i ordonnancée FP/DP activée en t_i dans une période active de niveau $PG_i(t_i)$.

Propriété 7 *Le pire temps de réponse d'une tâche* τ_i *ordonnancée* FP/DP *activée en* t_i *dans une période active de priorité* $PG_i(t_i)$ *est obtenu dans le scénario pire cas où toutes les tâches potentiellement plus prioritaires que* τ_i *sont synchrones au début de la période active de niveau de priorité* $PG_i(t_i)$. *La première activation de* τ_i *est en* $t_i^0 = t_i \ mod \ T_i$ *tel que* $0 \leq t_i^0 \leq T_i - 1$ *(cf. figure 6)*

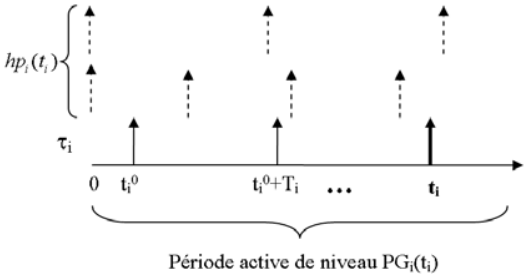


FIG. **6.** *Scénario pire cas préemptif pour* DP

Pour trouver le pire temps de réponse d'une tâche τ_i, il faut donc est tester tous les instants t_i du type $t_i^0 + kT_i, k \in \mathcal{N}$ dans [0,L[, où t_i^0 est un instant initial d'activation dans une période active, tel que $0 \leq t_i^0 < T_i$.

Le lemme suivant permet de réduire le nombre de points à tester et montre une certaine analogie avec l'ordonnancement FP. En déterminant la durée maximale d'une période active de niveau $PG_i(t_i)$ initiée en t_i^0.

Lemme 1 *Le pire temps de réponse d'une tâche* τ_i *est obtenu pour une tâche ordonnancée à l'instant* $t_i = t_i^0 + k \cdot T_i$, *avec* $t_i^0 \in [0, T_i - 1]$, $k \in \mathcal{N} \cap [0, K]$ *et* K *est le plus petit entier tel que* $t_i^0 + (K+1) \cdot T_i \geq \mathcal{B}_i(t_i^0)$, *où :*

$$\mathcal{B}_i(t_i^0) = \sum_{\tau_j \in hp_i \cup sp_i(t)} \left\lceil \frac{\mathcal{B}_i(t_i^0)}{T_j} \right\rceil \cdot C_j + \left\lceil \frac{\mathcal{B}_i(t_i^0) - t_i^0}{T_i} \right\rceil \cdot C_i.$$

On retrouve l'analogie avec la période active de niveau P_i pour l'ordonnancement FP mais pour un ensemble de scénarii d'activation plus grand (T_i scénarii d'activation pour τ_i qui démarrent en t_i^0, avec $t_i^0 \in [0, T_i - 1]$).

-Scénario pire cas DP ou FP/DP non préemptif :

Par analogie, le scénario pire cas pour maximiser le temps de réponse d'une tâche τ_i activée en t_i dans une période active de priorité $PG_i(t_i)$ avec un ordonnancement DP ou FP/DP en contexte non préemptif est celui du context préemptif en considérant un effet non préemptif en début de période active priorité $PG_i(t_i)$. Cet effet non préemptif est maximum lorsque l'on place en -1 une tâche de durée maximale dans $\overline{hp}_i \cup \overline{sp}_i(t)$.

Déterminer le pire temps de réponse d'une tâche préemptive ou non préemptive τ_i ordonnancée par un algorithme DP ou FP/DP respectant la propriété 2 est donc obtenu en calculant le temps de réponse obtenu pour τ_i activée à instant t_i borné par L dans une période active de niveau $PG_i(t_i)$. Le calcul du lemme 1 doit être adapté pour tenir compte de l'effet non préemptif comme suit ([25]) :

$$\mathcal{B}_i(t_i^0) = \sum_{\tau_j \in hp_i \cup sp_i(t)} \left\lceil \frac{\mathcal{B}_i(t_i^0)}{T_j} \right\rceil \cdot C_j + \left\lceil \frac{\mathcal{B}_i(t_i^0) - t_i^0}{T_i} \right\rceil \cdot C_i + \max^*_{\tau_k \in \overline{hp}_i \cup \overline{sp}_i(t)} (C_k - 1).$$

Ce lemme est appliqué dans la section 5 pour le calcul du pire temps de réponse d'une tâche ordonnancée FIFO, EDF et FP/FIFO.

5. Conditions de faisabilité

5.1. Conditions de faisabilité FP

Si r_i est le pire temps de réponse d'une tâche τ_i alors une CNS de faisabilité est : $\forall i = 1 \dots n, r_i \leq D_i$ et $U \leq 1$. Cette CNS est valable en contexte préemptif et non préemptif. Nous rappelons maintenant les différents résultats qui permettent de déterminer si un jeu de tâches respecte ses contraintes temporelles.

Ordonnancement FP préemptif :
Condition de faisabilité générale ([20], [39])
Cette condition de faisabilité consiste à calculer dans la première période active de niveau P_i les instants de fin d'exécution successifs $W_i(t)$ de τ_i, activée aux instants t dans $0, T_i, 2T_i, \dots \left\lfloor \frac{L_i}{T_i} \right\rfloor$ (cf. section 4.2).

Théorème 1 *Le pire temps de réponse d'une tâche τ_i ordonnancée* FP *est solution de $r_i = max_{t \in S}(W_i(t) - t)$ où $W_i(t)$ est solution de : $W_i(t) = (1 + \left\lfloor \frac{t}{T_i} \right\rfloor)C_i + \sum_{\tau_j \in hp_i} \left\lceil \frac{W_i(t)}{T_j} \right\rceil C_j$ et $S = \{kT_i, k = 0 \dots K, k \in \mathcal{N}\}$ où K est tel que $W_{i,KT_i} \leq (K+1)T_i$*

$W_i(t)$ est l'instant de fin d'exécution de la tâche τ_i activée dans une période active de niveau P_i (cf. 4.2). La tâche τ_i activée en KT_i se termine avant sa prochaine requête activée en $(K+1)T_i$, elle vérifie $W_{i,KT_i} \leq (K+1)T_i$. L'instant W_{i,KT_i} est donc un instant oisif de niveau P_i, il n'y a plus de tâches de priorité supérieure ou égale à τ_i activées avant W_{i,KT_i} car si il y en avait, τ_i activée en KT_i ne serait pas terminée en W_{i,KT_i}. Nous avons donc $W_{i,KT_i} = L_i$.

Cas particuliers :

- Dans le cas particulier où $\forall i, D_i \leq T_i$, on impose que la tâche τ_i activée en 0 dans son scénario pire cas soit terminée au plus tard en $D_i \leq T_i$. Nous avons donc d'après le théorème précédent $K = 0$ et $W_{i,0} = L_i$. Le temps de réponse pire cas de τ_i est alors solution de : $r_i = W_{i,0} = C_i + \sum_{\tau_j \in hp_i} \left\lceil \frac{W_{i,0}}{T_j} \right\rceil C_j$, formule proposée par [15].
- Dans le cas particulier où $\forall i, D_i = T_i$, avec l'ordonnancement RM, [23] proposent une CS de faisabilité pour un jeu de tâches sporadiques ou périodiques : $U \leq n(2^{1/n} - 1)$. Ce qui conduit à garantir la faisabilité des tâches au minimum lorsque la charge du processeur est inférieure à 73%.

Ordonnancement FP non préemptif :
Une première approche pour le calcul des temps de réponse pires cas est de reprendre le calcul des temps de réponse en contexte préemptif en ajoutant l'effet non préemptif maximum. Cette approche conduit à des conditions de faisabilité suffisantes mais non nécessaires, en prenant en compte potentiellement des tâches plus prioritaires que τ_i alors que τ_i a débuté son exécution.

Nous nous intéressons dans la suite aux conditions de faisabilité nécessaires et suffisantes ([10], [11]). Cette condition de faisabilité consiste à calculer dans la première période active de niveau de priorité P_i les instants de début d'exécution successifs $\overline{W}_i(t)$ de τ_i, activée aux instants t dans $0, T_i, 2T_i, \dots \left\lfloor \frac{L_i}{T_i} \right\rfloor$ (cf. section 4.2).

Théorème 2 *Le pire temps de réponse d'une tâche τ_i ordonnancée* FP *est solution de $r_i = max_{t \in S}(\overline{W}_i(t) + C_i - t)$ où $\overline{W}_i(t)$ est solution de : $\overline{W}_i(t) = \left\lfloor \frac{t}{T_i} \right\rfloor C_i + \sum_{\tau_j \in hp_i} (1 + \left\lfloor \frac{W_i(t)}{T_j} \right\rceil)C_j + max^*_{\tau_k \in \overline{hp}_i}(C_k - 1)$ et $S = \{kT_i, k = 0 \dots K, k \in \mathcal{N}\}$ où K est tel que $\overline{W}_i(KT_i) + C_i \leq (K+1)T_i$*

On remarquera qu'à la différence de cas préemptif,

on compte les tâches plus prioritaires dans un intervalle fermé (dans $[0, W_i(t)]$) et que seules les instances de τ_i activée avant t sont prises en compte.

5.2. Conditions de faisabilité DP

5.2.1 Conditions de faisabilité pour EDF

Il existe deux possibilités de résoudre un problème de faisabilité d'un jeu de tâches sporadiques ou périodiques ordonnancées EDF. La première approche consiste à utiliser la demande processeur h(t) (cf. 2.4). La seconde se base sur le calcul du pire temps de réponse d'une tâche. Le pire temps de réponse pire cas peut également servir à calculer un temps de réponse de bout en bout. La complexité du test basé sur h(t) est la plus faible.

Tests de faisabilité d'EDF en contexte préemptif -Test basé sur $h(t)$

[3] proposent une CNS de faisabiltié basée sur h(t) en contexte préemptif. Ils montrent qu'une CNS pour la faisabilité d'un jeu de tâches sporadiques ou périodiques est de vérifier dans le scénario pire cas synchrone pour toute les tâches τ_i telles que $D_i \leq t$, que $h(t) \leq t$. Si par hypothèse le scénario synchrone est un scénario pire cas pour la faisabilité des tâches avec EDF, alors il n'est pas nécessaire de tester que $h(t) \leq t$ au delà de L (cf. 4.1). En effet, le scénario qui redémarre au plus tôt en L est soit synchrone, donc déja testé soit asynchrone et donc pas le pire cas. En ne prenant en compte que les instants de variation de $h(t)$ aux échéances absolues de tâches, nous obtenons le théorème suivant :

Théorème 3 *Une CNS de faisabilité pour* EDF *en contexte préemptif est :* $\forall t \in S, h(t) \leq t$, *où* $S = \sqcup_{i=1}^{n}\{kTi + D_i, k \in \mathcal{N}\} \cap [0, L[$

Cas particulier :

- Lorsque $\forall i = 1 \ldots n, D_i \geq T_i$ [3] montrent alors qu'il y a une stricte équivalence entre le théorème 3 et la condition $U \leq 1$. Dans ce contexte, $U \leq 1$ est donc une CNS de faisabilité pour EDF préemptif. Ce résultat avait déja également été exprimé par [23] dans le cas $\forall i = 1 \ldots n, D_i = T_i$.

-Calcul de r_i pour EDF préemptif

Soit t_i, l'instant d'activation d'une tâche τ_i dans une période active de niveau $PG_i(t_i)$. Nous avons $PG_i(t_i) = P_i(t_i, t_i) = t_i + D_i$

Ce calcul se base sur le scénario pire cas pour l'ordonnancement DP déterminé dans la section 6. Avec EDF, nous avons $hp_i(t_i) = \{\tau_j, j \neq i / D_j \leq t_i + D_i\}$ et $\overline{hp}_i(t_i) = \{\tau_j, / D_j > t_i + D_i\}$.

Les tâches qui peuvent géner τ_i activée en t_i sont :

- les tâches τ_j dans $hp_i(t_i)$ dont l'échéance est inférieur ou égale à celle de τ_i (d'échéance absolue $t_i + D_i$) activée en t_i. Pour une tâche τ_j d'échéance D_j, il y en a au maximum $1 + \left\lfloor \frac{t_i + D_i - D_j}{T_j} \right\rfloor$, contrairement à l'ordonnancement FP où une tâche plus prioritaire peut toujours retarder τ_i si τ_i n'est pas terminée.
- les activations précédentes de τ_i. Il y en a au maximum $1 + \left\lfloor \frac{t_i}{T_i} \right\rfloor$

Pour le calcul du pire temps de réponse maximum EDF préemptif, on retrouve le même principe de calcul du point fixe FP à la différence qu'il faut tester plus de points qu'avec FP, où seuls les instants d'activation de τ_i dans le scenario synchrone sont à tester. Avec EDF, il faut en principe tester tous les instants $t_i \in \cup\{\{t_i^0 + kT_i, k \in \mathcal{N}, 0 \leq t_i^0 < T_i\} \cap [0, L[$.

[35] est à l'origine de ce résultat. Le théorème en propose une présentation différente légèrement optimisée.

Théorème 4 *Le pire temps de réponse d'une tâche τ_i ordonnancée* EDF *est donné par :* $r_i = \max_{t \in \mathcal{S}_i}\{W_i(t) - t\}$ *où :*

$W_i(t) = \sum_{\tau_j \in hp_i(t)} min(\left\lceil \frac{W_i(t)}{T_j} \right\rceil, 1 + \left\lfloor \frac{t + D_i - D_j}{T_j} \right\rfloor) \cdot C_j + (1 + \left\lfloor \frac{t}{T_i} \right\rfloor) \cdot C_i$

and $\mathcal{S}_i = \bigcup_{t_i^0=0}^{T_i - 1} \mathcal{S}_i(t_i^0)$, *avec* $\mathcal{S}_i(t_i^0)$ *l'ensemble des instants t tels que $t = t_i^0 + k \cdot T_i$ tels que :*

- $t_i^0 \in [0, T_i - 1[$
- $k \in \mathbb{N} \cap [0, K]$, *avec K le plus petit entier tel que :* $t_i^0 + (K+1)T_i \leq B_i(t_i^0)$
- *et t_i^0 appartient à une période active de niveau de priorité $PG_i(t_i^0)$.*

Remarque: $B_i(t_i^0)$ est calculé suivant la formule donnée dans la section 4.2 pour DP en contexte préemptif.

EDF en contexte non préemptif
-Test basé sur $h(t)$

[10] et [11] étendent le test basé sur $h(t)$ et montrent qu'il faut en cela tenir compte à un instant t de toutes les tâches synchrones ayant au moins une échéance dans l'intervalle $[0, t]$ (dans $h(t)$) et de l'effet non préemptif dû à une tâche moins prioritaire au sens EDF qui placée à l'instant 0 aurait une échéance $D_k > t$, de durée maximale (par convention, cette quantité est nulle si il n'y a pas de tâche moins prioritaire).

Théorème 5 *Une CNS de faisabilité pour* EDF *en contexte non préemptif est :* $\forall t \in S, h(t) +$

$max^*_{\tau_k, D_k > t}\{C_k - 1\} \leq t$, *où* $S = \sqcup_{i=1}^{n}\{kTi + D_i, k \in \mathcal{N}\} \cap [0, L[$

Cas particulier : ([14])
Dans le cas particulier où $\forall i, D_i = T_i$, le test se simplifie comme suit :

Lemme 2 *Une CNS de faisabilité pour* EDF *en contexte non préemptif est :* $\forall t \in S, h(t) + max^*_{\tau_k, D_k > t}\{C_k - 1\} \leq t$, *où* $S = \sqcup_{i=1}^{n}\{kTi + D_i, k \in \mathcal{N}\} \cap [0, max_{j=1...n}(D_j)[$

En effet, au delà de $max_{j=1...n}(D_j)$, l'effet non préemptif disparaît et l'équivalence $h(t) \leq t$ et $U \leq 1$ est vérifiée comme pour le cas préemptif.

-Calcul de r_i pour EDF non préemptif
Le calcul du temps de réponse EDF non préemptif se fait sur les mêmes instants qu'en préemptif. Il faut cependant tenir compte de l'effet non préemptif du aux tâches dans $\overline{hp_i}(t)$. De plus, on calcule l'influence des tâches plus prioritaires que τ_i dans $[0, \overline{W}_i(t)]$.

Théorème 6 *Le pire temps de réponse d'une tâche* τ_i *ordonnancée* EDF *non préemptif est donné par :* $r_i = \max_{t \in \mathcal{S}_i}\left\{\overline{W}_i(t) - t + C_i\right\}$ *où :*
$\overline{W}_i(t) =$
$$\sum_{\tau_j \in hp_i(t)} (1 + \left\lfloor \frac{\min(t + D_i - D_j; \overline{W}_i(t))}{T_j} \right\rfloor) C_j + \left\lfloor \frac{t}{T_i} \right\rfloor C_i + max^*_{\tau_j \in \overline{hp}_i(t)} (C_j - 1)$$
et $\mathcal{S}_i = \bigcup_{t_i^0=0}^{T_i-1} \mathcal{S}_i(t_i^0)$, *avec* $\mathcal{S}_i(t_i^0)$ *l'ensemble des instants* $t = t_i^0 + k \cdot T_i$ *tels que :*

- $t_i^0 \in [0, T_i[$
- $k \in \mathbb{N} \cap [0, K]$, *avec* K *le plus petit entier tel que :* $t_i^0 + (K+1)T_i \geq B_i(t_i^0)$

Remarque: $B_i(t_i^0)$ est calculé suivant la formule donnée dans la section 4.2 pour DP en contexte non préemptif.

5.2.2 Conditions de faisabilité FIFO

Dans cette section, nous rappelons deux résultats concernant l'ordonnancement en monoprocesseur. Le premier résultat est valide pour tout ordonnanceur. Le second s'applique à l'ordonnancement FIFO.
Avec FIFO, la priorité d'une tâche est sa date d'activation. Toute tâche activée au plus tard en t_i est donc plus prioritaire que τ_i. Nous avons donc $PG_i(t_i) = P_i(t, t_i) = t_i$ et $hp_i(t) = \{\tau_j, j \neq i\}$

Lemme 3 *Le temps de réponse* $r_{i,t}$ *d'une tâche* τ_i *activée en* t *dans une période active de niveau* $PG_i(t)$ *est :* $W_{i,t} = (1 + \left\lfloor \frac{t - t_i^0}{T_i} \right\rfloor) C_i + \sum_{\tau_j \in hp_i(t)} (1 + \left\lfloor \frac{t - t_i^0}{T_i} \right\rfloor) C_j$

Ce lemme peut être simplifié en remarquant que $(1 + \left\lfloor \frac{t - t_i^0}{T_i} \right\rfloor) C_i = (1 + \left\lfloor \frac{t}{T_i} \right\rfloor) C_i$ car $0 \leq t_i^0 < T_i$. Or t est borné par L (cf. 4.1). De plus, seuls les instants multiples d'une date d'activation d'une tâche dans le scénario synchrone font évoluer $r_{i,t}$. Il vient le théorème donnant le temps de réponse FIFO d'une tâche :

Théorème 7 *Le pire temps de réponse* r_i *d'une tâche* τ_i *est solution de* $r_i = max_{t \in S} \sum_{\tau_j \in hp_i(t)} (1 + \left\lfloor \frac{t}{T_j} \right\rfloor) C_j - t$

Pour des tâches sans gigue, ce théorème se simplifie en ([10]) :

Propriété 8 *Une condition nécessaire et suffisante de faisabilité d'un ensemble* $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ *de* n *tâches sporadiques ou périodique ordonnancées selon* FIFO, *est que : (i)* $\sum_{j=1..n} C_j \leq min_{j=1..n}(D_j)$ *et (ii) que le facteur d'utilisation* U *vérifie* $U \leq 1$.

6. Conditions de faisabilité FP/FIFO

Dans cette partie, nous nous intéréssons à l'ordonnancement FP/FIFO en contexte préemptif et non préemptif. Les tests de faisabilité sont basé sur la calcul du pire temps de réponse.
La prise en compte de la politique DP=FIFO locale permet d'améliorer les conditions de faisabilité et des trouver des solutions alors que FP n'en trouve pas ([25]). Cela vient du fait que les conditions de faisabilité établies dans la section 5.1 sont valable pour tout algorithme DP. Elles sont donc plus restrictives.
Le lecteur intéréssé par l'étude de FP/EDF en contexte préemptif et non préemptif peut consulter respectivement [12] et [25, 27].
En ce qui concerne FP/FIFO, nous avons :

- $sp_i(t) = \{\tau_j \in sp_i,\ 0 \leq t\}$;
- $\overline{sp}_i(t) = \emptyset$;

Nous donnons maintenant les conditions de faisabilité FP/FIFO en contexte preemptif et non preemptif.

Théorème 8 *Le pire temps de réponse d'une tâche* τ_i *ordonnancée* FP/FIFO *en contexte préemptif*

est : $r_i = \max_{t \in \mathcal{S}_i} \{W_i(t) - t\}$*, où :*

$$W_i(t) = \sum_{\tau_j \in hp_i} \left\lceil \frac{W_i(t)}{T_j} \right\rceil \cdot C_j + \sum_{\tau_j \in sp_i(t) \cup \tau_i} \left(1 + \left\lfloor \frac{t}{T_j} \right\rfloor \right) \cdot C_j$$

et $\mathcal{S}_i = \bigcup_{t_i^0=0}^{T_i-1} \mathcal{S}_i(t_i^0)$*, avec* $\mathcal{S}_i(t_i^0)$ *l'ensemble des instants tels que* $t = t_i^0 + k \cdot T_i$ *tels que :*

- $t_i^0 \in [0, T_i[$
- $k \in \mathbb{N} \cap [0, K]$*, avec* K *le plus petit entier tel que :*
  $t_i^0 + (K+1) \cdot T_i \geq B_i(t_i^0)$

Théorème 9 *Le pire temps de réponse d'une tâche* τ_i *ordonnancée* FP/FIFO *en contexte non préemptif est :* $r_i = \max_{t \in \mathcal{S}_i} \{W_i(t) - t\} + C_i$*, où :*

$$W_i(t) = \sum_{\tau_j \in hp_i} \left(1 + \left\lfloor \frac{W_i(t)}{T_j} \right\rfloor \right) \cdot C_j + \sum_{\tau_j \in sp_i(t)} \left(1 + \left\lfloor \frac{t}{T_j} \right\rfloor \right) \cdot C_j + \left\lfloor \frac{t}{T_i} \right\rfloor \cdot C_i + max^*_{\tau_k \in \overline{hp_i}} (C_k - 1),$$

et $\mathcal{S}_i = \bigcup_{t_i^0=0}^{T_i-1} \mathcal{S}_i(t_i^0)$*, avec* $\mathcal{S}_i(t_i^0)$ *l'ensemble des instants tels que* $t = t_i^0 + k \cdot T_i$ *tels que :*

- $t_i^0 \in [0, T_i[$
- $k \in \mathbb{N} \cap [0, K]$*, avec* K *le plus petit entier tel que :*
  $t_i^0 + (K+1) \cdot T_i \geq B_i(t_i^0)$

Nous illustrons l'amélioration obtenue en terme de faisabilité en considérant trois tâches : τ_1, τ_2 and τ_3 ordonnancées FP/FIFO en contexte non préemptif par un calcul de région d'ordonnançabilité. Cet exemple est issu de [27]. Les tâches τ_2 et τ_3 partagent la même priorité. τ_1 est la tâche la plus prioritaire :

- $T_1 = 50,\ D_1 = 100,\ P_1 = 1$;
- $T_2 = 10,\ D_2 = 25,\ \ P_2 = 2$;
- $T_3 = 10,\ D_3 = 35,\ \ P_3 = 2.$

La figure 7 illustre l'amélioration obtenue en terme de région d'ordonnançabilité avec FP/FIFO par rapport à FP. Les axes de gauche et de droite représentent respectivement C_1 et C_2. L'axe vertical représente la différence pour un C_1 et un C_2 donné entre la valeur de C_3 limite pour la faisabilité du jeu de tâches avec FP/FIFO et celle obtenue avec FP. La différence est toujours positive ou nulle.

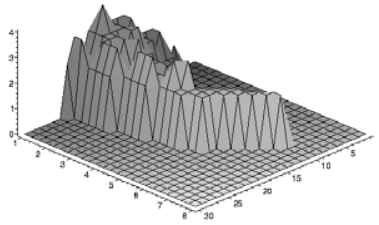


FIG. **7. Amélioration de la région d'ordonnançabilité avec** FP/FIFO

7. Conclusion

Dans cet article, nous avons étudié les différents résultats disponibles dans l'état de l'art pour le dimensionnement d'un système temps réel monoprocesseur en contexte préemptif et non préemptif. Nous avons précisé les différents modèles qui permettent de caractériser un problème d'ordonnancement. Nous avons étudié les principaux algorithmes d'ordonnancement issus de l'état de l'art en analysant les contextes pour lesquels ils sont optimaux. Nous avons étudié la notion de période active pour les politiques d'ordonnancement FP, DP et FP/DP. Cette notion nous a permis ensuite de déterminer les scenarii pires cas ainsi que les conditions de faisabilité pour le respect de contraintes temporelles. Nous avons étudié dans cet article les politiques FP, EDF et FP/FIFO.

Références

[1] S. Aldarmi and A. Burns. Time-cognizant value functions for dynamic real-time scheduling. *Technical Report YCS-306, Real-Time Research Group, Department of Computer Science*, The University of York, U.K.

[2] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Dept. Comp. Science Report YCS 164, University of York*, 1991.

[3] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems*, Vol. 2, pp. 301-324, 1990.

[4] S. Baruah, A. K. Mok, and L. Rosier. Preemptively scheduling hard real-time sporadic tasks on one processor. *Proceedings of the 11th Real-Time Systems Symposium*, pp. 182-190, 1990.

[5] A. Burns, D. Prasad, A. Bondavalli, et al. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, Vol. 46, p. 305-325, 2000.

[6] A. Burns and A. Wellings. Engineering a ahrd real-time system : From theory to practive. *Software Practice and Experience*, 25(7), pp. 705-726, 1995.

[7] G. Buttazo and J. Stankovic. RED : A Robust Earliest Deadline Scheduling. *3rd international Workshop on responsive Computing*, Sept 1993.

[8] A. Demers, S. Keshav, and S. Sker. Analysis and simulation of a fair queueing algorithm. *Journal of ternetworking Research and experience*, pages 3-26, October 1990.

[9] M. Dertouzos. Control Robotics : the procedural control of physical processors. *Proceedings of the IFIP congress*, pp. 807-813, 1974.

[10] L. George. Ordonnancement en-ligne temps réel critique dans les systèmes distribués. *Thèse de doctorat en informatique*, Université de Versailles St-Quentin, 26 janvier 1998.

[11] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive scheduling real-time uniprocessor scheduling. *INRIA Research Report*, No. 2966, September 1996.

[12] M. González Harbour and J. C. Palencia. Response time analysis for tasks scheduled under EDF within Fixed Priorities. *IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[13] J. Jackson. Scheduling a production line to minimize maximum tardiness. *Tech. rep. University of California*, Report 43, 1955.

[14] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *IEEE Real-Time Systems Symposium*, pp. 129-139, San Antonio, USA, December 1991.

[15] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Comp. Jour.*, 29(5), pp. 390-395,, 1986.

[16] D. Katcher, J. Lehoczky, and J. Strosnider. Scheduling models of dynamic priority schedulers. *Research Report CMUCDS-93-4*, Carnegie Mellon University, Pittsburgh, April 1993.

[17] G. Koren and D. Shasha. D-over : An Optimal Online Scheduling Algorithm for over loaded real-time system. *INRIA Research Report*, No. 138, Feb 1992.

[18] J. Y. Le Boudec and P. Thiran. A note on time and space methods in network calculus. *Technical Report, No. 97/224*, Ecole Polytechnique Fédérale de Lausanne, Suisse, April 1997.

[19] G. Le Lann. A methodology for designing and dimensioning critical complex computing systems. *IEEE Intl. symposium on the Engineering of Computer Based Systems*, pp 332-339, March Friedrichshafen, Germany, 1996.

[20] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proceedings 11th IEEE Real-Time Systems Symposium*, pp 201-209, Dec. Lake Buena Vista, FL, USA, 1990.

[21] J. Y. T. Leung and M. Merril. A note on premptive scheduling of periodic, Real Time Tasks. *Information Processing Letters.Vol 11*, num 3, Nov. 19980.

[22] J. Liebeherr, D. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE ACM Transactions on Networking*, Vol. 4, No. 6, pp. 885-890, December 1996.

[23] L. C. Liu and W. Layland. Scheduling algorithms for multi-programming in a hard real time environment. *Journal of ACM*, Vol. 20, No 1, pp. 46-61, January 1973.

[24] C. Locke. Best effort decision making for real-time scheduling. *PhD thesis*, Computer science department, Carnegie-Mellon university, 1986.

[25] S. Martin. Maîtrise de la dimension temporelle de la qualité de service dans les réseaux. *Ph.D. Thesis*, University of Paris 12, July 2004.

[26] S. Martin, P. Minet, and L. George. End-to-end response time with Fixed Priority scheduling : Trajectory approach vs. holistic approach. 2004.

[27] S. Martin, P. Minet, and L. George. Improving non-preemptive Fixed Priority scheduling with Dynamic Priority as secondary criterion. *RTS'05*, Paris, France, April 2005.

[28] J. Migge, A. Jean-Marie, and N. N. Timing Analysis of Compound Scheduling Policies : Application to Posix1003.1b. *Journal of Scheduling, Kluwer Academic Publishers*, vol. 6, Num. 5, pp457-482, 2003.

[29] A. K. Mok. Fundamental design problems for the hard real-time environments. *MIT Ph.D. Dissertation*, May 1983.

[30] P. Muhlethaler and K. Chen. A Scheduling Algorithm for Taks Described by Time Value Functions. *Real Time Systems*, Volume 10, number 3, May 1996.

[31] N. Navet. *Evaluation de performances temporelles et optimisation de l'ordonnancement de tâches et messages*. Thèse de doctorat de l'Institut National Polytechnique de Lorraine, Novembre 1999.

[32] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks : the multiple node case. *IEEE ACM Transactions on Networking*, Vol. 2, No. 2, April 1994.

[33] P. Puschner and A. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1) :67–91, Jul. 1997.

[34] B. Sprunt, L. Sha, and P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1) :27–60, June 1989.

[35] M. Spuri. Analysis of deadline scheduled real-time systems. *INRIA Research Report*, No. 2772, Jan. 1996.

[36] D. B. Stewart and P. K. Khosla. Real-time scheduling of sensor-based control systems. 8^{th} *IEEE workshop on real-time operating systems*, Atlanta, USA, May 1991.

[37] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44, 1, January 1995.

[38] K. Tindell, A. Burns, and A. J. Wellings. An extendible Approach For Analysing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems 6(2)v*, 1994.

[39] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, Vol. 9, pp. 147-171, 1995.

[40] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessors and Microprogramming, Euromicro Journal*, Vol. 40, 1994.

Analyse des temps de réponse et de la demande processeur en ordonnancement temps réel de tâches périodiques

Pascal Richard
Laboratoire d'Informatique Scientifique et Industrielle
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 - 1 avenue Clément Ader
BP 40109
86961 Chasseneuil cedex
pascal.richard@ensma.fr

Abstract

La mise en œuvre de nombreux systèmes temps réel repose sur un exécutif temps réel. Il fournit un ordonnanceur à priorité pour attribuer le processeur aux tâches durant la vie du système. La validation d'un système temps réel revient alors à vérifier que l'ordonnancement respecte les contraintes temporelles des tâches pour tous les comportements possibles du système. Nous présentons deux techniques d'analyse d'ordonnançabilité des tâches périodiques à échéances strictes : l'analyse du temps de réponse et l'analyse de la demande processeur. Les principes de base de ces méthodes, leurs distinctions et leurs généralisations pour analyser des systèmes complexes sont présentés à travers des exemples simples dans le contexte de systèmes temps réel monoprocesseurs. Nous mettrons enfin en évidence les deux principaux problèmes à étudier durant la conception de nouveaux tests d'ordonnançabilité.

1 Introduction

Une tâche périodique est activée à intervalle régulier de temps. La longueur de cet intervalle définit la *période* d'activation de la tâche. Chaque occurence d'une tâche est appelée une *requête* (instance ou job). Dans un système temps réel dur, chaque requête est soumise à une échéance temporelle stricte. Le non respect d'une contrainte temporelle n'est pas admissible à l'exécution. L'ordonnancement de tâches périodiques est associé aux systèmes de contrôle-commande (p. ex., lecture échantillonnée d'un signal, etc.) et aux systèmes réactifs en général. La périodicité de l'activation des tâches est l'unique spécificité de l'ordonnancement temps réel par rapport à la théorie de l'ordonnancement classique. Cette nuance peut paraître mineure, mais en général la périodicité des tâches change la nature combinatoire des problèmes d'ordonnancement. Toutefois, l'ordonnancement temps réel nécessite de recourir aux mêmes techniques algorithmiques de résolution de problèmes combinatoires [20].

Le concepteur d'un système temps réel définit un ensemble (ou système) de tâches périodiques. Elles sont soumises à des contraintes temporelles dont le respect doit être validé (vérifié) avant la mise en service du système ou durant l'exécution des tâches si leurs caractéristiques ne sont pas connues a priori. La validation et l'ordonnancemement d'un système sont très souvent considérés comme des problèmes disjoints, bien qu'ils sont bien sûr fortement imbriqués en réalité. Lorsque l'ordonnancement est construit durant l'exécution (c-à-d., en-ligne), alors la validation consiste à analyser le comportement d'un algorithme fondé en général sur l'attribution de priorités aux tâches. L'ordonnancement et la validation sont alors clairement séparés et n'intéragissent pas (c.à.d., pas d'échange de données entre les étapes de validation et d'ordonnancement). Cette décomposition du problème d'ordonnancement consistant d'une part, à choisir un algorithme d'ordonnancement, et d'autre part, à analyser son comportement, permet de reduire la validation à un problème d'évaluation des performances. L'algorithme de validation est désigné sous le nom de *test d'ordonnançabilité* dans la littérature [3]. Un test positif indique que pour tous les comportements possibles du système alors les contraintes temporelles des tâches seront respectées.

Les tâches périodiques sont exécutées indéfiniment. Bien que l'ordonnancement soit infini, la périodicité des tâches permet de limiter la recherche des fautes temporelles dans un intervalle de temps borné, appelé intervalle de faisabilité ou d'étude (*feasibility interval*) [12]. La périodicité de l'ordonnancement est égale au ppcm des périodes des tâches et est désignée sous le nom d'*hyperpériode*. Simplifier le test

d'ordonnançabilité en un problème d'évaluation de performance suggère l'utilisation de techniques de simulation. Bien que cette technique soit très utile durant la conception préliminaire d'un système, nous rejettons son utilisation comme test d'ordonnançabilité pour les stratégies d'ordonnancement en-ligne. En effet, l'idée couramment rencontrée, consistant à simuler le comportement du système sur les bases des caractéristiques de l'ordonnanceur et du comportement pire cas de chaque tâche ne conduit pas nécessairement au pire comportement du système. L'instabilité d'un ordonnancement, liée à la variation des paramètres des tâches durant leur exécution, peut engendrer des situations qui n'auront jamais été simulées. La simulation, aussi bien que les tests logiciels classiques, ne permettent en aucun cas de conclure avec certitude que les tâches respecteront leurs contraintes temporelles. Seule une analyse pire cas, qui repose sur la caractérisation du pire comportement de l'application temps réel, permettra d'arriver à une conclusion avec un temps de calcul raisonnable.

Dans le premier exposé sur l'ordonnancement dans le cadre de l'école d'été ont été présentés les algorithmes et leurs propriétés de base [9]. Afin d'éviter les répétitions, nous conseillons aux lecteurs débutants en ordonnancement temps réel de lire ce premier article. Dans la suite nous considérons la validation d'un système reposant sur un ordonnanceur en-ligne. Trois techniques d'analyse pire cas existent dans la littérature (tests d'ordonnançabilité) : *l'analyse du facteur d'utilisation du processeur, l'analyse de la demande processeur et l'analyse du temps de réponse.* Ces techniques ne sont pas équivalentes. Dans ce second volet sur l'ordonnancement, nous nous limitons aux deux dernières techniques.

La majorité des monographies en temps réel présentent séparément l'ordonnancement des tâches à priorité fixe et à priorité dynamique. Nous choisissons ici une présentation différente afin de clairement dissocier la différence entre les tests exacts et approchés, comment ces méthodes se généralisent et enfin montrer les difficultés à surmonter pour constuire de nouveaux tests. Nous présentons paragraphe 2 la démarche générale de ces deux techniques tests et deux fonctions fondamentales pour calculer la demande processeur générée par les tâches. Le paragraphe 3 présente les tests exacts et le suivant les tests approchés. Enfin, dans le paragraphe 5, nous présentons quelles sont les difficultés à surmonter pour définir un nouveau test et les hypothèses simplificatrices couramment utilisées dans la littérature.

Nous utiliserons les notations suivantes : la première date de réveil de la tâche τ_i est notée r_i, sa pire durée d'exécution C_i, l'échéance relative à l'arrivée sera notée D_i et la période entre deux activations est T_i. Pour les systèmes à priorité fixe, nous considérons que les tâches sont triées selon leur priorité (τ_1 est la tâche la plus prioritaire). Nous nous limitons volontairement à l'ordonnancement monoprocesseur. La généralisation (de l'analyse du temps de réponse) aux systèmes distribués revient en pratique à valider les processeurs séparément en introduisant une gigue sur les activations des tâches comme variables de liaison entre les systèmes monoprocesseurs qui composent les systèmes distribués. Nous renvoyons aux références bibliographiques [16, 17] pour plus de détails.

2 Présentation des deux analyses

Dans le principe, l'analyse du temps de réponse et l'analyse de la demande processeur repose sur un même constat : *un dépassement d'échéance ne survient jamais lorsque le processeur est libre.* Ces analyses se restreignent à l'étude des intervalles de temps où le processeur exécute des tâches : *les périodes d'activité (busy periods).* L'analyse du temps de réponse et l'analyse de la demande processeur portent sur l'étude d'une ou plusieurs périodes d'activité. Chaque période d'activité à analyser va être caractérisée par *un scénario pire cas.* Bien que les deux analyses reposent sur le même constat, les principes de ces deux tests sont différents.

Définition 1 *L'analyse du temps de réponse est un test d'ordonnançabilité en deux étapes :*

- *tout d'abord le pire temps de réponse R_i (ou une borne supérieure) de chaque tâche est calculé,*
- *puis le respect des échéances est testé en vérifiant : $R_i \leq D_i$,$1 \leq i \leq n$ (complexité algorithmique en $O(n)$).*

L'analyse des temps de réponse se fait tâche par tâche et la complexité du test d'un système de tâches est principalement liée au calcul des pires temps de réponse. Par contre, l'analyse de la demande processeur est un test considérant toutes les tâches simultanément.

Définition 2 *L'analyse de la demande processeur revient à tester pour tout intervalle de temps $[t_1,t_2]$ que la durée maximum cumulée (ou une borne supérieure) des exécutions des requêtes qui ont leur réveil et leur échéance dans l'intervalle est inférieure à $t_2 - t_1$ (c-à-d., n'excède pas la longueur de l'intervalle).*

Nous présentons maintenant les concepts nécessaires à l'étude de ces deux techniques d'analyse de l'ordonnançabilité d'un système de tâches.

2.1 Périodes d'activité et scénario de test

Définition 3 *Une période d'activité du processeur est un intervalle de temps $]a,b[$ de l'ordonnancement tel que le processeur a exécuté toutes les requêtes arrivées avant la date a et a terminé à la date b toutes les requêtes arrivées à partir de la date a.*

Lorsque l'ordonnanceur est conservatif, c'est-à-dire qu'il n'insère pas de temps creux dans l'ordonnancement s'il existe une tâche prête à s'exécuter, le nombre de périodes d'activités différentes est fini puisque l'ordonnancement est périodique avec une période égale au ppcm des périodes des tâches. Précisément, l'ordonnanceur va être confronté exactement au même scénario de réveils des tâches à chaque début d'hyperpériode et prendra en conséquence exactement les mêmes décisions. Toutefois, l'analyse de toutes les périodes d'activité n'est en général pas possible puisque leur nombre est exponentiel. De plus, trouver dans quelle période d'activité une tâche ne respectera pas son échéance n'est pas un problème simple et nécessite dans la majorité des cas un temps de calcul exponentiel dans la taille du système de tâches à analyser. Pour des systèmes simples de tâches, nous allons présenter les résultats analytiques connus pour caractériser la période d'activité pour trouver les tâches ne respectant pas leurs échéances dans un système non ordonnançable.

Dans le cas d'un système à priorité fixe, tester l'ordonnançabilité d'une tâche τ_i ne nécessite pas de considérer les tâches moins prioritaires puisqu'elles n'engendreront pas d'interférence sur τ_i. Ceci revient à définir une période d'activité se limitant à un sous-ensemble de tâches prioritaires.

Définition 4 *(Systèmes à priorité fixe) Une période d'activité du processeur de niveau i est un intervalle de temps où le processeur n'exécute que des tâches ayant une priorité supérieure ou égale à i (level-i busy period).*

Une période d'activité va être caractérisée par les dates de réveils de requêtes qui la débutent. Nous pouvons maintenant définir la notion de scénario.

Définition 5 *Un scénario est l'ensemble des dates de réveils des requêtes permettant de caractériser* une *période d'activité du processeur.*

Deux cas se produiront suivant que les pires scénarios considérés durant le test se produiront ou non durant la vie du système :

- le scénario se produit nécessairement dans la vie du système : le test d'ordonnançabilité résultant sera alors exact et définit une condition nécessaire et suffisante pour que le système de tâches soit ordonnançable,
- le scénario ne se produit pas forcément dans la vie du système : le test sera alors approché puisque la demande processeur aura été surestimée. Ceci introduit donc du *pessimisme* dans le test d'ordonnançabilité, c'est-à-dire qu'une borne supérieure de la demande processeur est calculée. Le test sera uniquement une condition suffisante d'ordonnançabilité : si le test renvoie vrai alors le système est ordonnançable, sinon on ne peut pas conclure.

Exemple 1 *Quelques exemples de scénarios conduisant soit à un test exact, soit à un test approché :*

- *Tests exacts : le scénario se produit toujours dans la vie de l'application.*
  - *Ordonnancement de tâches périodiques à départ simultané et à priorité fixe : le pire scénario est défini par le réveil simultané d'une requête de chaque tâche (c-à-d., l'instant critique). Ce scénario se produit au démarrage de l'application.*
  - *Ordonnancement de tâches à départ différé et gigue sur activation. On peut toujours définir les valeurs des gigues de façon à créer un instant critique (resynchroniser l'activation des tâches au début d'une période d'activité) et se ramener ainsi au pire scénario présenté ci-dessus.*
- *Tests approchés : le scénario peut ne pas se produire dans la vie de l'application.*
  - *Ordonnancement de tâches périodiques à départ différé et à priorité fixe : le pire scénario est défini par le réveil simultané des tâches. Savoir si ce scénario se produira dans la vie de l'application est co-NP-Complet (problème de congruences simultanées). En conséquence, la demande processeur calculée sera une borne supérieure.*
  - *Ordonnancement de tâches périodiques à priorité fixe en non-préemptif : le pire scénario pour la tâche τ_i survient lorsqu'elle est réveillée en même temps que les requêtes des tâches plus prioritaires (τ_1,...,τ_{i-1}) et juste après de la plus longue tâche parmi les moins prioritaires ($\max_{j=i+1,...,} C_j$). Analyser si ce scénario surviendra ou non dans la vie de l'application n'est pas un problème simple a priori. En conséquence le test correspondant sera approché.*

Afin de faciliter la conception de tests exacts, deux hypothèses se rencontrent presque systématiquement dans la littérature. Elles permettent d'assurer que le pire scénario se produit dans la vie du système. Ces hypothèses sont d'introduire une *gigue sur activation* (permettant ainsi de traiter les systèmes distribués) ou des *tâches sporadiques* (la période entre deux requêtes est une durée minimum et non une durée exacte, comme dans le cas des tâches périodiques). Ces hypothèses supplémentaires permettent de considérer un problème d'ordonnancement plus général, mais dont les pires scénarios sont plus simples à caractériser.

Exemple 2 *Voici deux exemples introduisant une simplification en jouant sur la périodicité des activations*

des tâches :

- *Ordonnancement de tâches sporadiques à priorité fixe et à départ différé : le scénario où toutes les tâches se réveillent simultanément pourra se produire dans la vie du système. Dans le cas de tâches strictement périodiques, une telle caractérisation n'existe pas.*
- *Ordonnancement EDF de tâches sporadiques : le pire temps de réponse d'une tâche τ_i survient dans une periode d'activité où toutes les tâches autres que τ_i sont réveillées simultanément et où τ_i se réveille à une date d'échéance d'une autre tâche. Le test exact de Spuri [19] dans le cas sporadique ne serait pas exact si les tâches étaient strictement périodiques, car le pire scénario ne se produirait pas forcément dans la vie du système.*
- *Ordonnancement de tâches périodiques à priorité fixe et départ différé avec gigue sur activation des tâches : les gigues vont permettre de resynchroniser les tâches de façon à définir le pire scénario qui se produira dans la vie du système.*

2.2 Evaluation de la demande processeur

Caractériser l'activité du processeur revient à compter les requêtes activées dans un intervalle de temps (c-à-d., dans la période d'activité). Deux fonctions vont être particulièrement utiles par analyser l'activité du processeur associée aux exécutions des requêtes des tâches. Cette activité sera dans la suite désignée sous le nom de *demande processeur*. Deux fonctions permettent de définir la demande processeur sur un intervalle de temps [14, 2] :

- la demande processeur des tâches réveillées avant une date t , qui sera notée $rbf(t)$ (request bound function, parfois notée $G(t)$),
- la demande processeur des tâches devant se terminer avant la date t (l'échéance est avant ou à la date t), qui sera notée dbf (demand bound function, parfois notée $H(t)$)

La première fonction est particulièrement utile pour analyser les systèmes à priorité fixe, tandis que la seconde est utile pour analyser l'ordonnancement produit par EDF. Nous nous limitons aux tâches à échéance contrainte ($D_i \leq T_i$ - constrained deadline) et à départ différé (asynchronous) pour illustrer les définitions de ces deux fonctions.

2.2.1 La fonction de travail du processeur

La demande processeur des tâches réveillées dans l'intervalle de temps $[0,t[$ repose sur le nombre k de réveils d'une tâche τ_i dans cet intervalle de temps. Cela nécessite de connaître la dernière requête activée avant la date t. Les deux inégalités suivantes permettent de déterminer k :

$$r_i + (k-1)T_i < t \quad \Rightarrow \quad k < \frac{t-r_i}{T_i} + 1$$

$$r_i + kT_i \geq t \quad \Rightarrow \quad k \geq \frac{t-r_i}{T_i}$$

Nous devons de plus assurer $k \geq 0$. Ces inégalités seront respectées pour : $k = \max(0, \lceil \frac{t-r_i}{T_i} \rceil)$. Notons que les requêtes comptabilisées ne sont pas nécessairement terminées à la date t. La durée maximum cumulée de travail processeur associée aux réveils de τ_i sur l'intervalle de temps $[0,t[$ est notée rbf (request bound function) :

$$rbf(\tau_i,t) = \max\left(0, \left\lceil \frac{t-r_i}{T_i} \right\rceil\right) C_i$$

La fonction de travail du processeur sur l'intervalle $[0,t[$ tient compte des requêtes de toutes les tâches :

$$W(t) = \sum_{j=1}^{n} rbf(\tau_j,t) \tag{1}$$

Le travail processeur est une fonction en escalier. La droite affine $f(t) = t$ définit la capacité maximum de traitement du processeur (avec une vitesse unitaire). Ainsi, lorsque $W(t) = t$, alors à cette date, le processeur a terminé toutes les requêtes réveillées dans l'intervalle $[0,t[$.

Pour les sytèmes à priorité fixe, la fonction de travail $W_i(t)$ est la durée cumulée des tâches de priorité supérieure ou égale à i et réveillées dans l'intervalle $[0,t[$:

$$W_i(t) = \sum_{j=1}^{i} rbf(\tau_j,t) \tag{2}$$

2.2.2 La demande processeur

La fonction $dbf(t_1,t_2)$ (demand bound function) est la durée cumulée des requêtes dont la date de réveil et l'échéance sont dans l'intervalle $[t_1,t_2]$. Nous pouvons définir le nombre k de réveils d'une tâche τ_i. Pour déterminer les requêtes dont les échéances surviennent dans un intervalle $[0,t]$, nous identifions la dernière requête de la tâche τ_i avec les deux inégalités suivantes :

$$r_i + (k-1)T_i + D_i \leq t \quad \Rightarrow \quad k \leq \frac{t-r_i-D_i}{T_i} + 1$$

$$r_i + kT_i + D_i > t \quad \Rightarrow \quad k > \frac{t-r_i-D_i}{T_i}$$

Nous savons aussi que $k \geq 0$. Ces inégalités seront respectées pour la tâche τ_i : $k = \max\left(0, \lfloor \frac{t-r_i-D_i}{T_i} \rfloor + 1\right)$. Nous avons donc la demande processeur des requêtes

tâches	C_i	D_i	T_i	R_i
τ_1	2	10	10	2
τ_2	10	25	30	14
τ_3	55	100	120	119

TAB. **1 –.** *Système de tâches. La colonne R_i indique les temps de réponse des tâches avec un ordonnanceur à priorité fixe*

d'échéances inférieures ou égales t dans l'intervalle $[0,t[$:

$$dbf(0,t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - r_i - D_i}{T_i} \right\rfloor + 1\right) \times C_i \quad (3)$$

Exemple 3 *Considérons un système de trois tâches à départ simultané dont les paramètres sont indiqués dans le tableau 1. La colonne R_i donne les temps de réponse des tâches lorsqu'elles sont ordonnancées avec des priorités fixes. Nous constatons que la tâche τ_3 n'est pas ordonnaçable (alors qu'elle le serait avec une échéance égale à la période). La figure 1 présente les fonctions $W_3(t) = rbf(t)$ et $dbf(t)$.*

2.2.3 Généralisations

En raisonnant sur des inégalités simples comme nous l'avons fait dans les paragraphes précédents, nous pouvons généraliser les formules précédentes pour traiter des systèmes plus complexes, comme ceux introduisant par exemple des paramètres supplémentaires dans la définition des tâches comme la gigue sur activation ou des temps de blocage associés à l'attente devant les ressources critiques (c-à-d., le facteur de blocage). Il convient toutefois d'être vigilent afin d'assurer que les valeurs calculées par les fonctions rbf ou dbf sont des bornes supérieures de la demande processeur, car sinon le test d'ordonnançabilité correspondant ne serait pas correct. Ceci nécessite de caractériser *le(s) pire(s) scénario(s)* d'arrivée des requêtes engendrant la plus grande activité du processeur.

Nous détaillons maintenant séparément les tests exacts et les tests approchés.

3 Tests exacts

Nous considérons pour simplifier la présentation les tâches à échéance contrainte (c-à-d., $D_i \leq T_i$) et qui sont à départ simultané (synchronous).

3.1 Analyse du temps de réponse

Le temps de réponse d'une requête est la différence entre sa date de fin et sa date de réveil. Le pire temps de réponse d'une tâche est le plus grand temps de réponse de ses requêtes. L'ordonnancement étant périodique, le nombre de valeurs différentes des temps de réponse des requêtes d'une tâche est fini et calculable.

3.1.1 Ordonnancement à priorité fixe

Comme nous l'avons vu précédemment, le calcul pratique du temps de réponse nécessite toujours de caractériser le(s) scénario(s) d'arrivée des tâches conduisant au pire temps de réponse de la tâche étudiée, déterminer une fonction d'analyse de la durée cumulée des tâches correspondant au(x) pire(s) scénario(s). Nous illustrons le calcul du pire temps de réponse d'une tâche τ_i dans un système à priorité fixe [**?**].

Lemme 1 *(Scénario) Le pire temps de réponse d'une tâche τ_i survient d'une période d'activité de niveau i débutant par un instant critique.*

Théorème 1 *(Test) Le pire temps de réponse de τ_i est défini par le plus petit point fixe de l'équation :*

$$W_i(t) = t$$

L'algorithme correspondant a été présenté dans le premier exposé sur l'ordonnancement et pour cette raison nous ne le détaillons pas (voir [9]). La complexité algorithmique du calcul du temps de réponse d'une tâche est pseudo-polynomiale : $O(n \sum_{i=1}^{n} C_i)$. Puisque les pires temps de réponse des tâches sont calculés en séquence, la complexité asymptotique du test d'ordonnançabilité est la même que celle pour analyser une tâche. Notons que l'existence d'un algorithme polynomial est un problème ouvert.

Exemple 4 *Sur le système de tâches présenté dans le tableau 1, les temps de réponse sont calculés en 3 itérations maximum et les temps de réponse correspondant sont indiqués dans la colonne R_i de ce même tableau.*

3.1.2 Ordonnancement EDF

Contrairement au système à priorité fixe, le pire temps de réponse d'une tâche ordonnancée par EDF ne survient pas nécessairement dans la première période d'activité du processeur [19]. Le pire temps de réponse d'une tâche peut être calculé en construisant (par simulation) l'ordonnancement EDF. Mais l'algorithme résultant sera exponentiel. A notre connaissance, aucun algorithme polynomial ou pseudo polynomial n'est encore connu pour calculer le pire temps de réponse exact des tâches ordonnancées par EDF.

3.2 Analyse de la demande processeur

L'analyse de la demande processeur est un test d'ordonnançabilité consistant à vérifier que toutes les requêtes devant s'exécuter dans tout intervalle de temps ne dépassent pas la capacité du processeur (c-à-d., la longueur de l'intervalle considéré). Contrairement

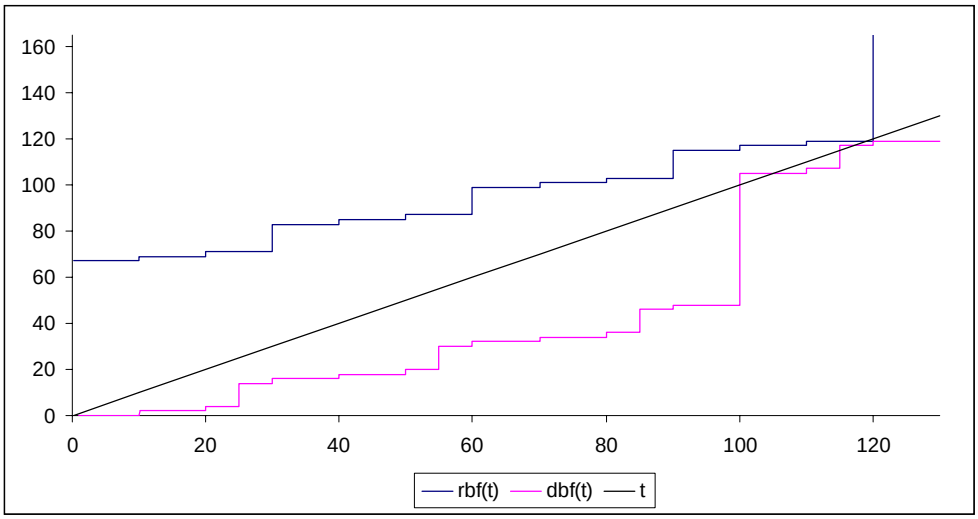


FIG. **1 –.** *Fonctions $rbf(t)$, $dbf(t)$ et $f(t) = t$ pour le système de tâches du tableau 1*

à l'analyse du temps de réponse, l'analyse de la demande processeur analyse toutes les tâches simultanément.

3.2.1 Ordonnancement à priorité fixe

Nous présentons dans la suite les principes de cette analyse à travers l'exemple des tâches à priorité fixe et à départ simultané [11].

Lemme 2 *(Scénario) Il est suffisant d'analyser l'intervalle de temps $[0,D_i)$ pour savoir si τ_i respectera ou non son échéance.*

Théorème 2 *(Test de Lehoczky, Sha et Ding) Dans un système de tâches à départ simultané, une tâche τ_i est ordonnançable si, et seulement si, il existe un instant $t \in (0,D_i]$ tel que $W_i(t) \leq t$.*

Cela revient donc à rechercher la valeur minimum de la fonction $W_i(t)/t$ dans l'intervalle $[0,D_i]$. La fonction de la demande processeur ne change de valeur qu'à des instant précis car l'équation 2 est une fonction en escalier. Le test va se limiter aux valeurs correspondant à des minima locaux de la fonction de la demande processeur. L'ensemble de ces valeurs définissent l'*ensemble des points d'ordonnancement* (Testing Set ou Scheduled Point Test) :

$$S_i = \{bT_j | j = 1 \ldots i, b = 1 \ldots \lfloor D_i/T_j \rfloor\} \qquad (4)$$

Ainsi, vérifier que la tâche τ_i est ordonnançable nécessite de calculer : $\min_{t \in S_i} \left(\frac{W_i(t)}{t} \right) \leq 1$. En conséquence, si une date $t \in S_i$ satisfait $W_i(t) \leq t$ alors τ_i est ordonnançable et il est inutile d'examiner d'autres points d'ordonnancement. Ainsi en pratique, seulement un sous-ensemble des points d'ordonnancement de S_i sera analysé. Mais, d'un point de vue complexité algorithmique, le nombre d'itérations du test est borné par le ratio: D_i/T_j. Donc, sa complexité algorithmique est pseudo-polynomiale. Le test complet se formule de la façon suivante :

$$\max_{i=1\ldots n} \left\{ \min_{t \in S_i} \left(\frac{W_i(t)}{t} \right) \right\} \leq 1$$

Exemple 5 *Nous illustrons ce test sur l'analyse de la tâche τ_3 du système de tâches présenté dans le tableau 1. Les ensembles de tests sont :*

$$\begin{aligned} S_1 &= \{10\} \\ S_2 &= \{10,20,30\} \\ S_3 &= \{10,20,30,40,50,60,70,80,90,100\} \end{aligned}$$

Les calculs de $W_3(t)/t$ pour l'ensemble S_3 sont présentés dans le tableau 2. Le système est non ordonnançable puisqu'aucune date $t \in S_3$ ne conduit à vérifier la condition : $W_3(t)/t \leq 1$.

Nous renvoyons à [14] pour une présentation des algorithmes et une variante de ce test (permettant d'avoir une complexité indépendante des valeurs des paramètres, mais exponentielle - $O(n2^n)$). On pourra aussi consulter [5] pour avoir des informations complémentaires.

3.2.2 Ordonnancement EDF

Les tâches étant à départ simultané, la plus grande charge processeur survient au démarrage de l'application. Précisémment, la demande processeur dans le cas de tâches à départ simultané vérifie [4] :

Lemme 3 *(Scénario)*

$$dbf(0,t_2 - t_1) \leq dbf(t_1,t_2) \qquad \forall t_1,t_2 \quad et \quad t_1 \leq t_2$$

t	10	20	30	40	50	60	70	80	90	100
$W_1(t)/t$	0,2									
$W_1(t)/t$	1,2	0,7	0,5							
$W_3(t)/t$	6,7	3,4	2,3	2,0	1,7	1,4	1,4	1,2	1,14	1,1

TAB. **2 –.** *Exécution du test associé au théroème 2 pour analyser le système de tâches du tableau 1.*

Nous pouvons simplifier l'expression du calcul de la demande du processeur puisque nous supposons $r_i = 0$ et $D_i \leq T_i$, $1 \leq i \leq n$:

$$dbf(0,t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \quad (5)$$

$$= \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \quad (6)$$

Ceci conduit à définir un intervalle d'étude pour EDF débutant à l'instant critique 0, jusqu'à $H = ppcm(T_i)$, puisque l'ordonnancement est périodique de période H. En relachant la contrainte d'intégralité de l'équation 6, il est démontré dans [4] que l'intervalle d'étude peut se limiter dans le cas $U < 1$ à l'intervalle $[0,t_{lim}]$, avec :

$$t_{lim} = \frac{U}{1 - U} \max_{i=1..n} (T_i - D_i)$$

Ceci permet d'établir le test d'ordonnançabilité suivant :

Théorème 3 *(Test de Baruah, Howell et Rosier) Un système de tâches périodiques et à départ simultané est ordonnançable avec un facteur d'utilisation $U < 1$ si, et seulement si :*

$$dbf(0,t) \leq t \qquad \forall t, 0 < t < t_{lim}$$

Exemple 6 *Nous illustrons le fonctionnement de ce test sur le système de tâches du tableau 1. Le facteur d'utilisation du processeur est $U = 0{,}99$ et ceci conduit à la valeur $t_{lim} = 2380$ (On remarque que le ppcm des périodes est égal à $H = 120$. Cette valeur peut être utilisée comme limite de temps puisque les tâches sont à départ simultané). Nous devons donc tester toutes les dates t entre 0 et H pour vérifier la condition $dbf(t) \leq t$. Si pour une valeur de t, cette inégalité n'est pas vérifiée alors le système est non ordonnançable. Par contre, si le test est positif pour toutes les dates t entre 0 et H alors le système est ordonnançable. Pour les dates $t \in [0,100[$, toutes les inégalités sont respectées. Par contre à la date $t = 100$, nous avons : $dbf(0,t) = 105 > t$ (nous pouvons aussi directement le constater sur la figure 1 : la courbe $dbf(0,t)$ passe au dessus de la droite représentant la capacité du processeur $f(t) = t$). Le système de tâches n'est donc pas ordonnançable sous EDF.*

Sous l'hypothèse U constant, la complexité de ce test est $O(n \max_{i=1..n}(T_i - D_i))$, l'algorithme est pseudo-polynomial. Ce test a été amélioré dans [18, 15], mais ces algorithmes sont eux aussi pseudo-polynomiaux. Notons que l'existence d'un algorithme fortement polynomial est un problème ouvert.

3.2.3 Généralisation

Les tests exacts présentés reposent sur l'analyse de la demande processeur, exploitent des propriétés permettant de limiter l'intervalle d'étude et enfin la localisent dans la première période d'activité du processeur (cas de tâches à départ simultané). En l'absence de caractérisation précise de la période d'étude, l'analyse de la demande processeur doit être généralisée afin d'établir un test d'ordonnançabilté. Ceci passe notamment par la définition plus générale de la fonction de la demande cumulée du processeur.

Définition 6 *Fonction de demande du processeur (Demand Bound Function). Soit τ_i une tâche, la fonction de demande du processeur $dbf(t_1,t_2)$ est la durée maximum cumulée d'exécution des tâches qui ont leurs réveils et échéances dans un intervalle de durée $[t_1,t_2]$.*

Cette définition de la fonction $dbf(t_1,t_2)$ permet de définir un test simple pour les tâches indépendantes, de façon analogue au paragraphe précédent. Un test général peut être formulé de la façon suivante :

Théorème 4 *Un système de tâches est ordonnançable si, et seulement si :*

$$\forall t_1 < t_2 \qquad dbf(t_1,t_2) \leq t_2 - t_1 \quad (7)$$

Remarquons que dans le cas général, pour montrer que le système est non ordonnançable il est suffisant de trouver une valeur de t telle que l'inégalité 7 ne soit pas satisfaite. La généralisation à des tâches dépendantes avec des structures conditionnelles a été faite dans [2].

Toutefois, toute généralisation soulève deux questions :

- Comment calculer efficacement la fonction dbf ?
- Comment choisir un ensemble de points d'ordonnancement aussi petit que possible et qui soit suffisant pour garantir la correction du test défini par l'équation 7?

4 Tests approchés

L'objectif d'un test approché est de fournir une *décision approchée* au problème d'ordonnançabilité :

si le test approché renvoie Ordonnançable, alors les tâches respecteront leurs contraintes temporelles, sinon on ne peut pas conclure. La principale motivation pour concevoir un test approché est de définir des algorithmes de test avec une complexité algorithmique plus faible, soit parce que le test exact nécessite un temps de calcul exponentiel ; soit parce que le test doit être utilisé en-ligne pour contrôler l'admission de nouvelles tâches périodiques.

Les tests approchés vont donc utiliser des valeurs approchées de la demande processeur (c-à-d.., des bornes supérieures des fonctions présentées dans le paragraphe 2) et limiter le nombre d'itérations nécessaires pour prendre une décision. Ces deux opérations vont introduire du *pessimisme* dans les méthodes d'analyse.

4.1 Tests approchés sans garantie

Il est important (et peu rassurant) de constater qu'aucune estimation quantitative du pessimisme des tests approchés n'est en général rigoureusement proposée dans la littérature. Les évaluations reposent uniquement sur des simulations numériques qui ne comparent pas le test approché avec un test exact. De façon générale, les tests approchés d'ordonnançabilité conduisent à surdimensionner les systèmes afin de satisfaire la condition suffisante d'ordonnançabilité définie dans le test de validation !

Exemple 7 *Considérons l'ordonnancement de tâches périodiques à priorité fixe, à départ simultané et en mode non préemptif. Nous avons défini le pire scénario dans l'exemple 1. Considérons deux tâches* τ_1,τ_2 *de périodes identiques* T *et de durées* C_1 *et* C_2*, des échéances* $D_1 = T/2$ *et* $D_2 = T$*. Le scénario va conduire aux pires temps de réponse :* $R_1 = R_2 = C_1+C_2$*. Puisque les tâches ont une période identique, alors la tâche* τ_2 *n'aura pas d'interférence avec* τ_1*. En conclusion le pire temps de réponse exact de* τ_1 *est* $R_1^* = C_1$*. Le rapport* $\frac{R_1}{R_1^*} = 1+\frac{C_2}{C_1}$ *et on constate que le pire temps de réponse associé au scénario du test peut être arbitrairement éloigné de la valeur exacte* R_1^**, puisque* C_2/C_1 *n'est borné par aucune constante. Le test classiquement utilisé pour valider des systèmes de tâches non-préemptifs (ou ordonnancement de messages dans un réseau CAN par exemple) est donc sans garantie de performance vis-à-vis d'un test exact.*

4.2 Tests approchés avec garantie

Nous rappelons tout d'abord quelques définitions sur l'approximation polynomiale, puis nous illustrons les tests avec garantie de performance sur les tâches à priorité fixe.

4.2.1 Approximation polynomiale

Une estimation du pessimisme des méthodes approchées peut être obtenue en utilisant des *algorithmes d'approximation* (ou approchés). Ces algorithmes sont utilisés pour résoudre de façon approchée des problèmes d'optimisation. L'intérêt d'un algorithme d'approximation est de posséder une *garantie de performance* vis-à-vis d'une méthode exacte. Précisément, soit A un algorithme approché et OPT une méthode exacte, alors la borne d'erreur ϵ $(0 < \epsilon < 1)$ de l'algorithme A pour toute instance I du problème d'optimisation est définie par :

$$\frac{|A(I) - OPT(I)|}{OPT(I)} \leq \epsilon$$

Un algorithme approché a une garantie de performance bornée par le *ratio* suivant : $r_A = 1 + \epsilon$ (pour un problème de minimisation). Ce ratio définit donc les pires résultats que pourra atteindre l'algorithme approché A en considérant toutes les instances possibles d'un problème d'optimisation. Une approximation polynomiale est un algorithme avec un ratio constant. Un schéma d'approximation est un algorithme paramétrique, de paramètre ϵ, qui peut s'approcher aussi près que possible de la valeur optimale de la fonction optimisée. Le ratio d'un schéma d'approximation polynomiale (PTAS - Polynomial Time Approximation Scheme) s'écrit sous la forme: $r_A \leq 1 + \epsilon$.

Un schéma d'approximation est complet (FPTAS - Fully Polynomial Time Approximation Scheme) s'il est un PTAS et que l'algorithme est en plus polynomial en fonction du paramètre $1/\epsilon$. Un FPTAS est le meilleur résultat d'approximation pouvant être obtenu pour résoudre un problème $\mathcal{NP}$-Difficile. Nous renvoyons à [8] pour des compléments sur la complexité et l'approximabilité des problèmes.

Depuis plusieurs années, les algorithmes d'approximation intéressent les concepteurs de tests afin de garantir les performances dans le pire cas des tests approchés. Toutefois, tester la faisabilité d'un système de tâches est un problème de décision, alors que les algorithmes approchés concernent les problèmes d'optimisation. Bien que le temps de réponse soit un critère quantitatif, il n'existe pas à notre connaissance de calcul de pire temps de réponse approché avec une garantie constante de performance par rapport au pire temps de réponse exact. Mais des tests approchés reposant sur l'analyse de la demande processeur ont été proposés.

4.2.2 Approximation des temps de réponse

Un test d'ordonnançabilité fournit une réponse binaire : *ordonnançable* ou *non ordonnançable*. Par contre, le temps de réponse est un critère quantitatif. Celui-ci peut donc être utilisé pour définir le ratio d'approximation du calcul du pire temps de réponse d'une tâche. A notre connaissance il n'existe pas de résultat dans la littérature de calcul d'un pire temps de réponse approché avec une garantie de performance par

rapport au pire temps de réponse exact. Nous donnons ci-après un premier résultat négatif d'approximation du temps de réponse.

Un moyen classique pour établir des bornes est de relacher la contrainte d'intégralité dans la fonction de la demande processeur (plus précisément dans les fonctions $rbf_i(0,t)$). Nous notons R_i^* la valeur exacte du pire temps de réponse :

$$\begin{aligned} R_i^* &\geq C_i + \sum_{j=1}^{i-1} \frac{R_i^*}{T_j} C_j \\ R_i^* &\leq C_i + \sum_{j=1}^{i-1} \left(1 + \frac{R_i^*}{T_j}\right) C_j \end{aligned}$$

En utilisant cette expression, on obtient une borne inféreure et une borne supérieure calculables en temps linéaire (c-à-d., $O(n)$) :

$$R_i^* \geq \frac{C_i}{1 - \sum_{j=1}^{i-1} \frac{C_j}{T_j}} = \overline{R}_i \quad (8)$$

$$R_i^* \leq \frac{\sum_{j=1}^{i} C_j}{1 - \sum_{j=1}^{i-1} \frac{C_j}{T_j}} = \tilde{R}_i \quad (9)$$

Nous montrons que ces bornes n'ont pas de garantie de performance: c'est-à-dire qu'elles peuvent être très éloignées des valeurs exactes des pires temps de réponse des tâches. Précisément, nous notons $\overline{R}_i$ (respectivement $\tilde{R}_i$) la borne supérieure du temps de réponse (respectivement la borne inférieure), alors les ratios $\overline{R}_i/R_i^*$ et $R_i^*/\tilde{R}_i$ tendent vers l'infini (d'un point de vue asymptotique).

Théorème 5 *Soit r le ratio $\overline{R}_i/R_i^*$:*

a. Le ratio n'est pas borné pour les systèmes de tâches quelconques.

b. Si nous supposons qu'il existe une constante K telle que $K > \sum_i (C_i)/\min_i C_i$, alors $r \leq K$.

Preuve : Nous montrons tout d'abord la première assertion. Considérons l'instance suivante avec deux tâches: $\tau_1(1-\epsilon,1,1)$ et $\tau_2(K\epsilon,K,K)$, où ϵ respecte $0 < \epsilon < 1$ et K est un entier supérieur à 1. Notons que les périodes sont proportionnelles, en conséquence RM sera optimal et une condition nécessaire et suffisante d'ordonnançabilité $C_1/T_1 + C_2/T_2 \leq 1$. Le facteur d'utilisation est:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1-\epsilon}{\epsilon} + \frac{K\epsilon}{K} = 1$$

Le système de tâches est donc ordonnançable et le pire temps de réponse exact est:

$$\begin{aligned} R_1^* &= \overline{R}_1 = 1 - \epsilon \\ R_2^* &= K \qquad \overline{R}_2 = \frac{1 + (K-1)\epsilon}{\epsilon} \end{aligned}$$

La borne $\overline{R}_2$ pour τ_2 conduit au ratio suivant :

$$\lim_{\epsilon \to 0} \frac{\overline{R}_2}{R_2^*} = \lim_{\epsilon \to 0} \frac{1}{K\epsilon} + \frac{(K-1)}{K} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} = \infty$$

Maintenant, si nous supposons qu'il existe une constante K telle que $K > \sum_i (C_i)/\min_i C_i$ pour tout système de tâches alors en partant des équations 9, nous pouvons écrire :

$$\frac{C_i}{1 - \sum_{j=1}^{i-1} \frac{C_j}{T_j}} \leq R_i^* \leq \frac{\sum_{j=1}^{i} C_j}{1 - \sum_{j=1}^{i-1} \frac{C_j}{T_j}}$$

ainsi,

$$\frac{\overline{R}_i}{R_i^*} \leq \frac{\sum_{j=1}^{i} C_j}{C_i} \leq K$$

Nous pouvons établir les mêmes types de résultat pour la borne inférieure du pire temps de réponse exact. L'existence d'algorithme approché (avec un ratio constant) pouvant être calculé en temps polynomial est un problème ouvert.

4.2.3 Approximation polynomiale de la demande processeur

Un test d'ordonnançabilité fondé sur l'analyse de la demande processeur n'est pas un problème d'optimisation, mais un problème de décision (c-à-d., qui retourne une valeur binaire). Toutefois, les techniques d'approximation vont pouvoir être utilisées, moyennant une adaptation de la définition de la garantie de performance. Nous présentons dans la suite le schéma d'approximation polynomiale de [1, 7] qui utilise le paramètre ϵ avec la sémantique suivante :

- si le test répond *ordonnançable* alors le système est ordonnançable quel que soit son comportement à l'exécution,
- si le test répond *non ordonnaçable*, alors il est non-ordonnançable avec certitude sur un processeur plus lent (avec la vitesse $1-\epsilon$). Mais sur un processeur de vitesse unitaire, aucune décision ne peut être prise.

Nous illustrons cette approche sur l'ordonnancement à priorité fixe [7]. La fonction $rbf(\tau_i,t)$ est une fonction en escalier non-décroissante. Le nombre de paliers dans cette fonction n'est pas borné polynomialement dans la taille du système à ordonnancer. Un moyen simple de définir un schéma d'approximation polynomiale est de ne considérer qu'un nombre borné k de paliers. Au delà, une fonction linéaire (continue) sera utilisée pour définir une borne supérieure de $rbf(\tau_i,t)$. Le nombre de paliers va être défini à l'aide du paramètre d'erreur ϵ :

$$k = \left\lceil \frac{1}{\epsilon} \right\rceil - 1$$

ϵ	0,01	0,1	0,2	0,3	0,4	0,5
k	101	11	6	5	4	3

TAB. **3 –.** *Nombre de paliers de la fonction $rbf(\tau_i,t)$ considérée avant la linéarisation de la fonction dans le test approché de Fisher et Baruah*

Nous pouvons maintenant définir l'approximation de la demande cumulée de la demande processeur de la tâche τ_i, qui sera notée $\overline{rbf}(\tau_i,t)$:

$$\overline{rbf}(\tau_i,t) = rbf(\tau_i t) \quad \text{Si} \quad t \leq (k-1)T_i \quad (10)$$

$$= C_i + t\frac{C_i}{T_i} \quad \text{Sinon} \quad (11)$$

La demande processeur approchée, est alors définie par :

$$\overline{W}_i(t) = C_i + \sum_{j=1}^{i-1} \overline{rbf}(\tau_i,t) \quad (12)$$

Pour terminer le test, Fisher et Baruah utilisent l'analyse de la demande processeur avec un ensemble de points d'ordonnancement (Testing Set) possédant un nombre polynomial d'entrées dans la taille du système de tâches à analyser et du paramètre de précision $1/\epsilon$:

$$\overline{S}_i = \{bT_j | j = 1 \ldots i-1, b = 1 \ldots k\} \quad (13)$$

Exemple 8 *Le tableau 3 donne la valeur de k en fonction de la borne d'erreur ϵ. Nous donnons figures 2 et 3, le graphique des fonctions $\overline{W}_2(t)$ et $\overline{W}_3(t)$ pour le système de tâches présenté dans le tableau 1 et des valeurs d'epsilon égales à 0,5 et 0,3.*

L'algorithme de test s'implémente très facilement en $O(n^2/\epsilon)$. Clairement, si ϵ est proche de 0, alors le nombre d'itérations effectuées par l'algorithme est très grand, mais le nombre d'itérations est polynomial en $1/\epsilon$. En conséquence, cet algorithme paramétrique est un FPTAS. Le choix du paramètre de précision ϵ est donc primordial pour obtenir un test rapide et avec une bonne garantie de performance.

Le cas des tâches avec des échéances arbitraires (c-à-d., telles que l'échéance D_i et la période T_i ne sont pas reliées par une contrainte) est présenté dans [7]. Par souci de concision, nous renvoyons à [1] pour l'approximation de la demande processeur pour EDF.

5 Difficulter pour concevoir les tests d'ordonnançabilité

Nous pensons que l'établissement d'un test d'ordonnançabilité devient difficile à concevoir lorsque l'algorithme d'ordonnancement n'est pas *robuste* (c-à-d., sujet aux anomalies d'ordonnancement) pour le problème considéré et que le problème d'ordonnançabilité est *difficile*. A notre connaissance, ces deux critères ne sont pas connus comme étant dépendants. Dans les deux cas, le calcul de la demande processeur (fonctions rbf ou dbf) doit donc reposer sur un scénario conduisant à la pire demande processeur. Un résultat analytique doit impérativement être construit pour garantir la correction du test.

Tâches	C_i	$D_i = T_i$	π_i
τ_1	1	3	1
τ_2	2	6	2
τ_3	4	12	3

TAB. **4 –.** *Tâches à échéance sur requête à ordonnancer sans préemption et avec des priorités fixes (π_i)*

5.1 Anomalies d'ordonnancement

Une anomalie d'ordonnancement est liée à l'instabilité de l'ordonnanceur. Ce problème, bien connu en environnement multiprocesseur, se formule de la façon suivante : *un système ordonnançable avec les pires durées d'exécution peut être non ordonnançable avec des durées d'exécutions plus petites.* Un ordonnanceur en-ligne sera dit *robuste* s'il n'est pas sujet à des anomalies d'ordonnancement pour le système de tâches considéré.

Exemple 9 *Voici quelques exemples de systèmes soumis à des anomalies d'ordonnancement en monoprocesseur :*

- *ordonnancement non préemptif (cas particulier d'ordonnancement avec ressources partagées en exclusion mutuelle),*
- *ordonnancement à priorité fixe avec contraintes de précédence,*
- *ordonnancement de tâches avec suspension (durant une opération d'entrée-sortie),*

Nous détaillons le premier exemple.

Exemple 10 *Nous présentons tableau 4 un système de tâches à ordonnancer sans préemption. La figure 4 présentent deux ordonnancements : le premier les tâches sont exécutées avec leurs pires durées d'exécution (a), tandis que dans le second la tâche 2 s'exécute en 1 seule unité de temps au lieu de 2 (b). Ainsi, réduire la durée d'exécution de τ_2 rend le système non-ordonnançable.*

5.2 Complexité

Déterminer la complexité du problème d'ordonnançabilité permet d'identifier quel type d'algorithme doit être construit pour analyser un système de tâches.

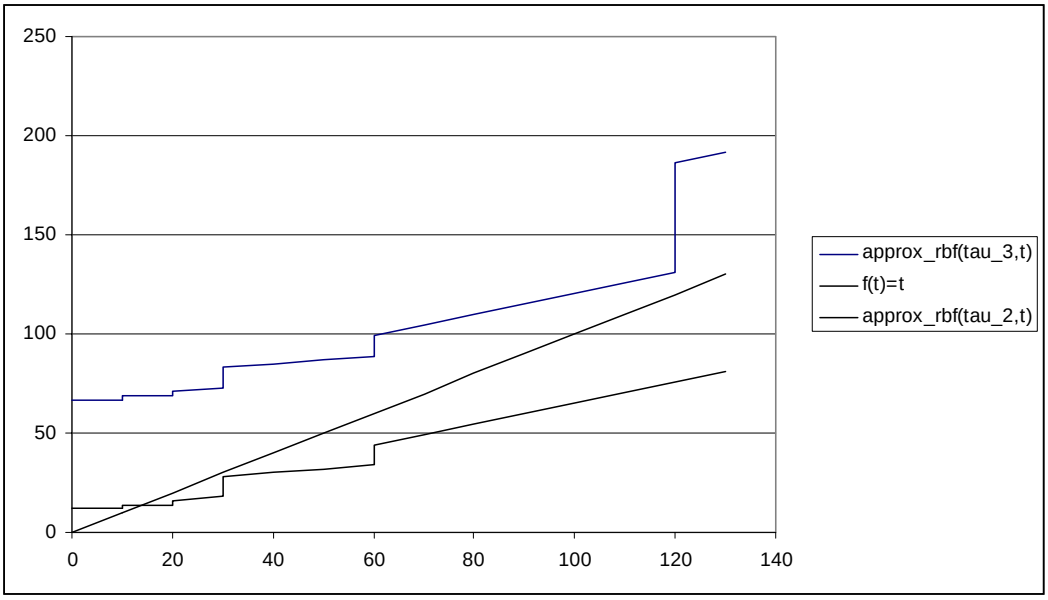


FIG. **2 –.** *Approximation de la demande processeur dans le test approché de Fisher et Baruah.* $\epsilon = 0{,}5$ $(k = 3)$*, le test conduit à l'ordonnançabilité de* τ_2*, mais* τ_3 *n'est pas ordonnançable sur un processeur de vitesse* $(1 - \epsilon)$*.*

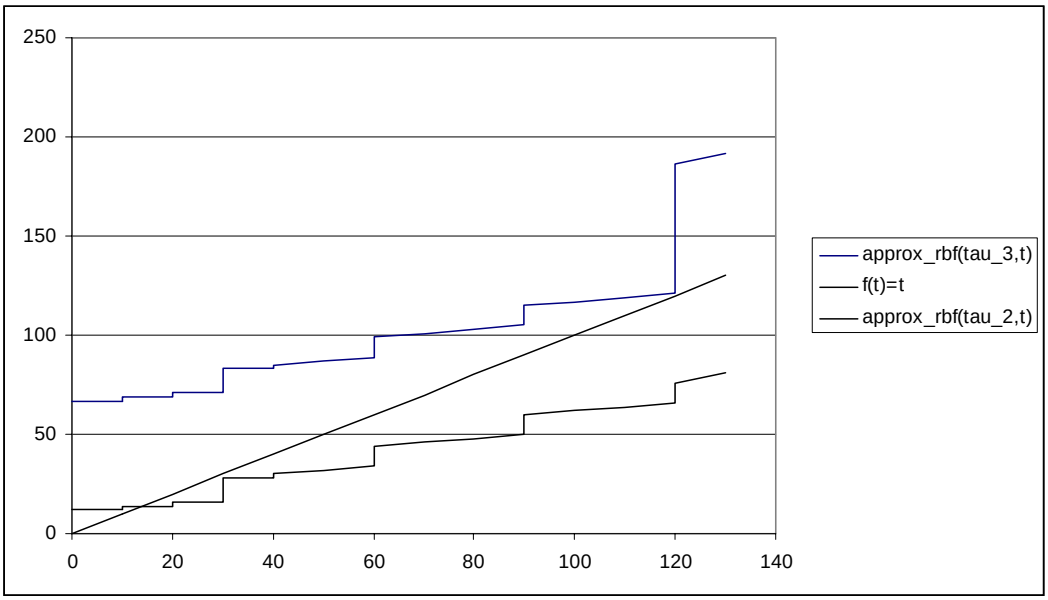
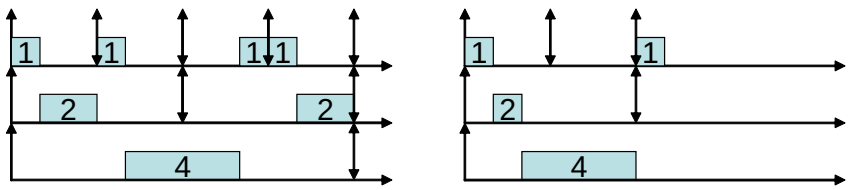


FIG. **3 –.** *Approximation de la demande processeur dans le test approché de Fisher et Baruah.* $\epsilon = 0{,}3$ $(k = 5)$*, le test conduit à l'ordonnançabilité de* τ_2*, mais* τ_3 *n'est pas ordonnançable sur un processeur de vitesse* $(1 - \epsilon)$*.*



a) $C_2 = 2$ *b)* $C_2 = 1$

FIG. **4 –.** *Anomalies d'ordonnancement en non-préemptif pour le système de tâches du tableau 4*

Si le problème est $\mathcal{NP}$-difficile alors les tests seront généralement des tests approchés car établir un test exact sera trop coûteux en temps de calcul et le(s) pire(s) scénario(s) seront difficiles à caractériser.

Le paysage est paradoxalement compliqué en ordonnancement de tâches périodiques puisque les problèmes sont soit dans $\mathcal{NP}$, soit dans co-$\mathcal{NP}$, ou bien non connus comme étant dans l'une de ces deux classes de problèmes.

Exemple 11 *Nous donnons une caractérisation simple de ces deux classes de problèmes vis-à-vis de l'ordonnançabilité des tâches à départ simultané et à échéance contrainte (les tests exacts ont été présentés plus haut) :*

*- pour les systèmes ordonnancés avec des priorités fixes, il est possible de décider en temps polynomial qu'*une tâche est ordonnançable *(par un algorithme non-déterministe). Le test de Lehoczky, Sha et Ding (voir [14]) permet de résoudre ce problème en temps pseudo-polynomial. Supposons qu'un algorithme non-déterministe (c-à-d., l'oracle d'une machine de Turing non déterministe) nous donne un point d'ordonnancement t et une tâche τ_i, alors la tâche est ordonnançable si la condition $W_i(t) \leq t$ est vraie. Cette vérification s'effectue en temps polynomial, montrant ainsi que le problème d'ordonnançabilité est dans $\mathcal{NP}$.*

*- on sait pour les systèmes ordonnancés par EDF décider en temps polynomial qu'*une tâche n'est pas ordonnançable *(par un algorithme non déterministe). Supposons qu'un algorithme non-déterministe nous donne un point d'ordonnancement t, alors la condition $dbf(0,t) > t$ permet de conclure que le système est non-ordonnançable en temps polynomial. Ceci établit que le problème d'ordonnançabilité est dans co-$\mathcal{NP}$.*

Il est assez surprenant que l'analyse d'ordonnançabilité pour un système de tâches donné est soit dans $\mathcal{NP}$, soit dans co-$\mathcal{NP}$, en fonction de l'algorithme d'ordonnancement considéré (à priorité fixe ou EDF). Après tout, cela laisse un peu d'espoir sur l'existence d'un test polynomial ! Les deux exemples considérés précédemment sont ouverts du point de vue de leur complexité (ils ne sont pas connus $\mathcal{NP}$-difficiles et aucun algorithme polynomial n'est connu).

Notons par ailleurs que l'ordonnancabilité des tâches en mode non-préemptif est $\mathcal{NP}$-Difficile au sens fort et que l'ordonnancement de tâches à départ différé est co-$\mathcal{NP}$-Complet au sens fort (y compris en préemptif).

6 Conclusion

Nous résumons ici les principales caractéristiques des analyses présentées dans cet article. L'analyse du temps de réponse s'étend assez facilement pour tenir compte de contraintes supplémentaires (ressources, systèmes distribués,...). Mais les tests correspondants sont généralement approchés et sans garantie de performance vis-à-vis d'un calcul exact des pires temps de réponse des tâches. Par contre, l'analyse de la demande processeur conduit à des algorithmes de tests performants, mais les résultats analytiques sur lesquels ils reposent sont moins faciles à étendre. Pour cette seconde technique d'analyse, des tests approchés avec garantie de performance permettent d'envisager leur utilisation pour contrôler l'admission en-ligne de nouvelles tâches périodiques. Toutefois, à notre connaissance, aucune généralisation de l'analyse de la demande processeur n'est connue pour les systèmes distribués.

Par terminer, nous avons volontairement limité la bibliographie aux références les plus récentes et avec des présentations pédagogiques. Des nombreuses pointeurs bibliographiques seront trouvés dans ces articles. En complément sur l'analyse de la demande processeur, nous conseillons en première lecture l'article [18] pour l'analyse EDF et l'article [14] pour l'analyse des systèmes à priorité fixe. Pour l'analyse du temps de réponse, nous renvoyons aux monographies bien connues du domaine [10, 6, 13].

Remerciements

Je tiens à remercier très sincèrement Stéphane Pailler et Frédéric Ridouard, doctorants dans notre laboratoire, pour leurs multiples lectures de cet article et leurs nombreux commentaires qui ont permis d'améliorer la présentation de cet article.

Références

[1] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. *Euromicro Int. Conf. on Real-Time Systems (ECRTS'04)*, 2004.

[2] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Journal of Real-Time Systems*, 24(1):93–128, 2003.

[3] S. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. *In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/CRC Press*, 2004.

[4] S. Baruah, L. Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 1:301–324, 1990.

[5] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed-priority systems. *IEEE Transactions on Computers*, 53(11): , nov 2004.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling, Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[7] N. Fisher and S. Baruah. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. *proc. Proceedings of the EuroMicro Conference on Real-Time Systems, (ECRTS'05)*, 2005.

[8] M. Garey and D. Johnson. *Computers and Intractability: a guide to the theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.

[9] L. George. Conditions de faisabilité pour l'ordonnancement temps réel. *Ecole d'Eté Temps Réel (ETR'05)*, 2005.

[10] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzales-Harbour. *A practitioner's handbook for real-time system analysis*. Kluwer Academic Publishers, 1993.

[11] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Extact characterization and average case behavior. *Real-Time Systems Symposium*, pages 166–171, 1989.

[12] J. Leung and M. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[13] J. W. S. Liu. *Real-Time Systems*, chapter Priority-Driven Scheduling of Periodics Tasks, pages 164–165. Prentice Hall, 2000.

[14] Y. Manabe and S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Journal of Real-Time Systems*, 14(2):171–181, 1998.

[15] M. Park and Y. Cho. Feasibility analysis of hard real-time periodic tasks. *Journal of Systems and Softwares*, 73:89–100, 2004.

[16] P. Richard. Analyse du temps de réponse des systèmes temps réel. *actes de l'Ecole d'Eté Temps Réel (ETR'03), Toulouse*, (1):241–262, 2003.

[17] P. Richard, M. Richard, and F. Cottet. Analyse holistique des systèmes temps réel distribués: principes et algorithmes. *in: ordonnancement pour l'informatique parallèle, Traité IC2, Hermès*, page , juin 2003.

[18] I. Ripoll, A. Crespo, and A. Mok. Improvement in feasibility testing for real-time tasks. *Journal of Real-Time Systems*, 11(1):19–39, 1996.

[19] M. Spuri. Analysis of deadline scheduled real-time systems. *INRIA Reseach Report 2772*, page 34p, 1996.

[20] J. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

Méthodes de calcul de WCET (Worst-Case Execution Time) État de l'art*

Isabelle Puaut
IRISA, Campus universitaire de Beaulieu,
35042 Rennes
puaut@irisa.fr

Abstract

La particularité des systèmes temps-réel strict est de devoir respecter de manière impérative des contraintes temporelles, qui sont le plus souvent des échéances de terminaison au plus tard. Dans de tels systèmes, il est courant d'utiliser des méthodes d'analyse d'ordonnancement, qui à partir de l'ensemble des tâches du système, déterminent si les échéances seront ou non respectées. La plupart de ces méthodes reposent sur la connaissance d'une borne supérieure du temps d'exécution de chaque tâche du système, nommée WCET pour Worst-Case Execution Time. *Cet article propose une synthèse des travaux effectués dans le domaine du calcul du WCET jusqu'à ce jour, et donne les tendances actuelles en terme d'obtention de WCET.*

1. Introduction

La particularité des systèmes temps-réel est d'avoir à respecter de manière plus ou moins impérative des contraintes temporelles. On distingue généralement les systèmes temps-réel *strict*, pour lesquels le non respect de ces contraintes peut avoir des conséquences catastrophiques, des systèmes temps-réel *souple* pour lesquels les contraintes sont définies pour assurer une qualité de service mais peuvent exceptionnellement être violées sans que cela ne soit dramatique. Cet article se place plus particulièrement dans le cadre des systèmes temps-réel strict.

Le temps d'exécution d'un programme dépend, en général, des valeurs des données en entrée du programme : ces valeurs déterminent un certain chemin d'exécution par le biais d'instructions de contrôle de flot. On appelle **temps d'exécution au pire cas** ou **WCET** (*Worst-Case Execution Time*) la valeur maximale de ce temps d'exécution pour l'ensemble des jeux de données en entrée possibles.

La connaissance du WCET d'une tâche peut être utile lors de la conception d'un système temps-réel, que ce soit au niveau du matériel, du système opératoire ou des applications :

– elle peut aider à dimensionner le matériel, qu'il s'agisse de déterminer si tel ou tel système est suffisamment puissant pour que les contraintes temps-réel d'une application puissent être garanties, ou bien d'estimer le nombre de tâches qu'un système donné peut supporter en assurant un respect de leurs échéances.
– dans le cadre du choix d'une politique d'ordonnancement pour un système temps-réel strict, l'analyse d'ordonnancement [2], réalisée hors-ligne, a pour but de vérifier que les échéances de toutes les tâches peuvent être satisfaites. Cette analyse est basée, entre autres, sur une estimation du WCET des différentes tâches.
– des informations sur le temps d'exécution de différentes parties d'un programme (boucles ou chemins critiques) peuvent être exploitées dans une démarche d'optimisation du code.
– la connaissance du WCET peut composer un élément d'argumentation pour la validation des systèmes embarqués, tels ceux de l'avionique, qui passent par un processus de qualification/certification.

Il est bien sûr indispensable que le WCET calculé soit supérieur (ou égal) au WCET réel, sinon il y a risque de violation de contraintes temps-réel, ce qui peut être fatal dans un contexte de temps-réel strict. Toutefois, pour être utile, le WCET ne doit pas être trop surestimé, ce qui conduirait à un surdimensionnement inutile du système.

Les méthodes habituellement mises en œuvre pour calculer le WCET se divisent en deux grandes catégories :

– les méthodes **dynamiques** consistent à mesurer le temps d'exécution du programme considéré sur un système réel ou sur un simulateur. Ces mesures doivent être réalisées pour tous les jeux d'entrées possibles, ou alors il faut être capable de définir un jeu d'entrées dont on est certain qu'il conduit au temps d'exécution le plus long.
– les méthodes **statiques** sont fondées sur une analyse statique du programme dans le but de s'affranchir des jeux d'entrée. Elles comportent en général deux phases ; l'analyse de *haut niveau* détermine tous les chemins possibles, et l'analyse de *bas niveau* évalue le temps d'exécution de chacun de ces chemins sur une architecture matérielle donnée.

Notons que les méthodes dynamiques permettent d'obtenir une valeur précise du WCET (si les mesures sont réalisées avec le jeu de test de pire cas, ce qui n'est pas toujours possible comme nous le montrons dans le paragraphe 2) tandis que les méthodes statiques conduisent à poser des hypothèses conservatrices pour des informations qui ne peuvent pas être connues précisément lors de l'analyse et calculent alors une borne supérieure du WCET (c'est-à-dire que le temps d'exécution maximum réel ne peut pas être supérieur au WCET calculé).

Dans cet article, nous donnons une vue d'ensemble des méthodes d'estimation des WCET. L'article est organisé de la manière suivante. Le paragraphe 2 est consacré aux méthodes d'évaluation dynamiques des WCET. Nous présentons les méthodes d'obtention des WCET par analyse statique dans leur globalité au paragraphe 3, et examinons plus en détail l'analyse de bas niveau dans le paragraphe 4. Nous concluons dans le paragraphe 5 par quelques réflexions sur les évolutions actuelles des méthodes de calcul de WCET.

Notons qu'une version plus détaillée de cet état de l'art est parue dans la revue Technique et Science Informatiques [1]. On se reportera à cet article pour plus détails concernant notamment l'analyse de bas-niveau et une comparaison des différentes méthodes d'obtention des WCET.

2. Méthodes dynamiques de détermination du WCET

Les méthodes dynamiques consistent à exécuter le programme dont on veut estimer le WCET (sur un système réel ou sur un simulateur) et à mesurer son temps d'exécution. Des jeux de tests pour l'exécution, qu'ils soient explicites (§ 2.1) ou symboliques (§ 2.2) sont nécessaires dans ce type de méthode.

2.1 Jeux de test explicites

Quel que soit le milieu de mesure (système matériel ou simulateur), il doit être alimenté par un code exécutable et par un jeu de données en entrée de ce programme. Pour mesurer le temps d'exécution au pire cas, il faut fournir, autant que faire se peut, le jeu de test qui conduit au temps d'exécution maximum. Se pose alors le problème de la définition de ce jeu de test. Plusieurs solutions sont envisageables :

- mesurer le temps d'exécution du programme pour *tous* les jeux d'entrées possibles : la durée du processus (temps de génération de tous les jeux de test, puis temps de mesure pour chacun d'entre eux) est, en général, rédhibitoire. Cependant, cette solution peut être retenue dans le cas de programmes simples, admettant peu de données en entrée, et pour lesquels le domaine de variation des entrées est limité.
- laisser le soin à l'utilisateur de définir le jeu de test pire cas : sa parfaite connaissance du programme peut lui permettre d'identifier ce jeu de test sans erreur. Là encore, cette solution est sans doute limitée à des applications très simples.
- générer automatiquement un jeu de test (ou un ensemble de jeux de test) qui conduit au temps d'exécution maximum. Quelques travaux basés sur des algorithmes évolutionnistes [3] ou sur l'algorithme du recuit simulé [4] ont été menés dans ce sens. Par exemple, dans [3], la génération des jeux de tests utilise un algorithme de type génétique, s'inspirant du principe d'évolution de Darwin. Une population d'*individus*, chacun représentant un jeu d'entrées du programme, est utilisée. Le principe de sélection des individus les *meilleurs* (ici, ceux produisant les plus grand temps d'exécution), ainsi que les lois biologiques de la reproduction entre individus (mutation, croisement) sont appliqués à la population de façon itérative. Ainsi au fur et à mesure de l'avancement de l'algorithme, de nouveaux individus qui répondent de mieux en mieux au problème (ici, maximiser les temps d'exécution) sont créés. Les estimations de WCET obtenues par ce type de méthodes ne sont pas complètement sûres dans la mesure où le jeu de test généré n'est pas garanti être celui du pire cas. L'intérêt de ces méthodes n'est toutefois pas négligeable puisqu'elles permettent de donner une limite inférieure au WCET et de compléter ainsi les calculs statiques qui, eux, en donnent une limite supérieure.

Ainsi, à moins d'être capable d'assurer une couverture exhaustive des chemins, la génération de tests explicites ne permet pas de garantir que le WCET obtenu est une borne supérieure des temps d'exécutions du programme (problème de sûreté). C'est pourquoi, nous ne détaillons pas plus avant ce type de méthode.

2.2 Jeux de test symboliques

Si l'on ne sait pas générer de jeu de test assurant un temps d'exécution de pire cas, une autre stratégie consiste à utiliser un jeu de test *symbolique*. Ce type de méthode est développé dans [5]. L'idée sous-jacente est que le point intéressant est mesurer les temps d'exécution de *tous* les chemins d'exécution possibles du programme, l'énumération de jeux d'entrée ne servant qu'à alimenter le programme de manière à parcourir tous ces chemins.

Le principe de cette méthode [5] est de poser l'hypothèse que les données en entrée sont inconnues et d'étendre le jeu d'instruction du processeur cible de sorte qu'il puisse réaliser des calculs à partir d'un ou plusieurs opérandes de valeur *inconnue* (par exemple, la somme d'un opérande de valeur connue et d'un opérande de valeur inconnue donne un résultat de valeur inconnue). Lorsqu'un branchement conditionnel est exécuté avec une condition de valeur inconnue, les deux chemins doivent alors être explorés. Ainsi, dans le cas de chemins d'exécution dépendant des données d'entrée, on explore *tous* les chemins possibles, assurant la sûreté de la méthode. Toutefois, si les programmes à analyser comportent des boucles, l'algo-

```
int impaire(int x) {
  int result;

  if (x % 2) {
    result = 1;
  } else {
    result = 0;
  }
  return(result);
}
```

```
void main() {
  int tab[4] = {34,45,12,5};
  int nb_impaires = 0;
  int nb_paires = 4;
  int i;

  for(i=0;i<4;i++) {
     nb_impaires += impaire(tab[i]);
     nb_paires -= impaire(tab[i]);
  }
}
```

FIG. **1 –.** *Code source d'un programme analysé (langage C)*

rithme d'exploration des chemins d'exécution mérite d'être soigneusement étudié pour éviter l'explosion combinatoire du nombre de chemins à évaluer. Une telle méthode ne peut bien entendu être mise en œuvre que dans le cadre d'un simulateur logiciel.

3. Détermination du WCET par analyse statique

Contrairement aux méthodes d'obtention des WCET par tests et mesures, présentées dans le paragraphe précédent, les méthodes d'obtention de WCET par analyse statique s'affranchissent des jeux d'entrée en opérant par analyse de la structure des programmes.

Les méthodes d'obtention du WCET par analyse statique opèrent par analyse de la structure des programmes, au niveau de leur code source et/ou objet. Elles peuvent être séparées en plusieurs composants logiques :

- l'*analyse de flot* qui, à partir du code source et/ou objet des programmes, détermine les chemins d'exécution possibles dans le programme ;
- l'analyse de *bas niveau* qui évalue l'impact de l'architecture matérielle sur le pire temps d'exécution du programme ;
- le *calcul* qui détermine le WCET à partir des résultats des autres analyses.

Nous nous concentrons dans ce paragraphe sur l'analyse de flot et le calcul de WCET. Le paragraphe 4 détaillera pour sa part l'analyse de bas niveau.

3.1 Analyse de flot

Nous développons ici les représentations du flot de contrôle les plus souvent utilisées par les analyseurs statiques de WCET (§ 3.1.1), puis la manière dont elles sont obtenues (§ 3.1.2).

3.1.1 Représentations du flot de contrôle

Blocs de base L'analyse du code objet des programmes est nécessaire pour l'analyse de bas niveau, car c'est à ce niveau que sont accessibles les informations sur la durée d'exécution des instructions. Presque toutes les méthodes d'analyse statique de WCET utilisent le découpage du code objet en *blocs de base*. Un bloc de base est une suite d'instructions purement séquentielle ne contenant qu'un seul point d'entrée et un seul point de sortie.

La figure 1 présente le code C d'un programme constitué de deux fonctions. La fonction principale `main` comporte une boucle, et pour chaque itération de la boucle la fonction `impaire` est appelée deux fois. Cette deuxième fonction comporte une structure conditionnelle. Ce programme nous servira à illustrer nos propos dans la suite de l'article. Par exemple, la figure 2 montre l'ensemble des blocs de base des fonctions *main* et *impaire*.

Graphe de flot de contrôle Un *graphe de flot de contrôle* décrit tous les enchaînements possibles entre blocs de base. La forme la plus utilisée de graphes de flot de contrôle est celle des graphes dont les nœuds sont des blocs de base et où les arcs représentent les relations de précédence entre blocs. La figure 2 donne le graphe de flot de contrôle de notre programme exemple, composé de dix blocs de base, numérotés de BB_1 à BB_{11}.

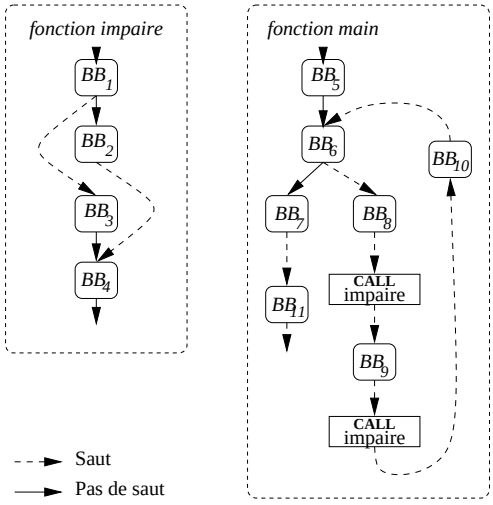


FIG. **2 –.** *Graphes de flot de contrôle du programme analysé*

On peut distinguer deux types d'arcs dans les graphes de flot de contrôle : les arcs "pas de saut" qui représentent l'exécution de deux blocs qui se suivent sans saut (absence de branchement, ou branchement non-pris) et les

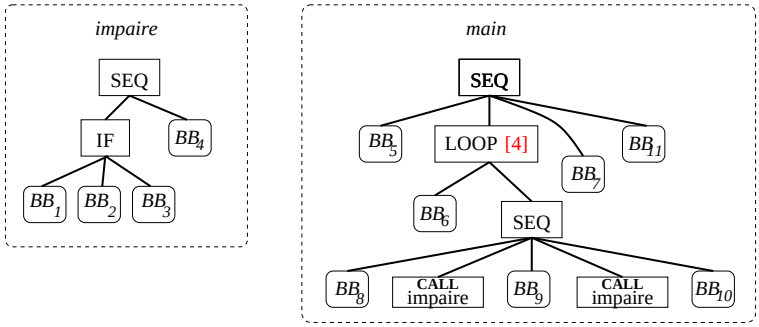


FIG. **3 –.** *Arbres syntaxiques des fonctions impaire et main*

arcs “saut” qui représentent les branchements pris (par exemple, arc de BB_{10} à BB_6). Cette catégorie de graphe de flot de contrôle est utilisée dans de nombreux travaux d’analyse statique [6, 7, 8].

Arbre syntaxique Un *arbre syntaxique* (*cf.* figure 3) est un arbre dont les nœuds représentent les structures du langage de haut niveau et dont les feuilles sont les blocs de base. Une représentation simple du programme ci-dessus par un arbre syntaxique peut être basée sur trois types de nœuds et deux types de feuilles.

- les nœuds de type SEQ possèdent au moins un fils. Ils représentent la mise en séquence de leurs sous-arbres fils.
- Les nœuds de type LOOP ont deux fils : le sous-arbre test et le corps de la boucle.
- les nœuds de type IF sont constitués d’un sous-arbre test et de deux sous-arbres “then” et “else”.
- les feuilles de type CALL représentent des appels de fonctions.
- enfin, les autres feuilles de l’arbre syntaxique sont les blocs de base du programme.

Informations supplémentaires sur le flot de contrôle Les représentations en blocs de base, graphes de flot de contrôle et arbre syntaxique ne sont pas suffisantes pour le calcul sûr et précis du WCET. En particulier, rien n’indique dans ces représentations que les chemins d’exécution sont de taille finie. Ces représentations doivent donc être complétées par des informations sur le comportement dynamique du code à analyser, de manière à restreindre le nombre de chemins d’exécution possibles, et donc le nombre de chemins à prendre en compte pour l’analyse. Ces informations sont le plus souvent utilisées pour :

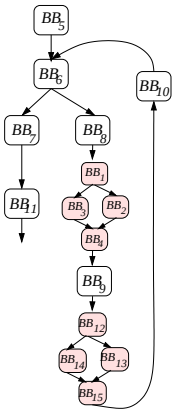
- Borner le nombre d’itérations des boucles. En effet, le WCET d’une boucle dépend non seulement de son code mais aussi de son pire nombre d’itérations. Cette information doit donc être associée aux boucles, et elle l’est le plus souvent sous forme d’une constante [9, 10]. Ainsi, la constante $[4]$ dans la figure 3 représente le pire nombre d’itérations de la boucle présente dans la fonction *main*.
- Indiquer la connaissance d’un chemin d’exécution dans une structure conditionnelle. Ceci est utile par exemple quand le choix d’une branche conditionnelle dépend d’un paramètre que l’on sait être constant.
- Restreindre le nombre de chemins d’exécution possibles en indiquant les chemins *infaisables* (chemins apparaissant dans le graphe de flot de contrôle du programme mais qui ne feront jamais partie d’une exécution réelle — par exemple, deux chemins d’exécution qui s’excluent mutuellement).

3.1.2 Obtention des informations de flot de contrôle

Le graphe de flot de contrôle est le plus souvent construit par analyse statique du code de bas niveau (code assembleur ou bytecode), et en utilisant soit un compilateur modifié [11], soit un outil dédié à la manipulation de code de bas niveau [12]. La représentation en arbre syntaxique, pour sa part, est obtenue par analyse statique d’un langage de haut niveau.

En ce qui concerne les informations supplémentaires sur le flot de contrôle, que nous avons énumérées dans le paragraphe 3.1.1, on peut distinguer deux possibilités pour les obtenir. La première fait appel à l’utilisateur (*i.e.* le programmeur) qui doit fournir ces informations en les ajoutant au code source sous forme d’annotations [9, 13, 14], ou interactivement [15]. La plupart des travaux concernant l’analyse statique de WCET utilisent ce type de méthodes.

La deuxième possibilité est de dériver ces informations automatiquement. Cette deuxième classe de méthodes ne permet pas d’obtenir ces informations de manière systématique. En effet, certains programmes ne se terminent pas, par exemple lorsqu’ils comportent des boucles sans fin, dont la sortie est assurée par une ou plusieurs instructions du corps de boucle, ce qui rend l’analyse trop complexe ; pour d’autres, le nombre maximum d’itérations des boucles peut aussi dépendre d’éléments extérieurs au programme, impossibles à identifier statiquement. Toutefois, dans le cas où ces méthodes sont utilisables, elles permettent d’éliminer les erreurs humaines dans le processus de mise en place des annotations. Les travaux décrits dans [16] utilisent



(a) Description du graphe

$$
\begin{aligned}
n_5 + n_{10} &= n_6 \\
n_6 &= n_7 + n_8 \\
n_7 &= n_{11} \\
n_8 &= n_1 \\
n_1 &= n_2 + n_3 \\
n_2 + n_3 &= n_4 \\
n_4 &= n_9 \\
n_9 &= n_{12} \\
n_{12} &= n_{13} + n_{14} \\
n_{13} + n_{14} &= n_{15} \\
n_{15} &= n_{10}
\end{aligned}
$$

(b) Contraintes du graphe

$$
\begin{aligned}
n_5 &= 1 \\
n_6 &\leq 4 \times n_5
\end{aligned}
$$

FIG. **4 –.** *Traduction d'un graphe de flot de contrôle en un système de contraintes*

l'analyse de flot de données pour identifier les nombres d'itérations des boucles ainsi que les chemins infaisables à l'intérieur de celles-ci. Ferdinand *et al.* [17] utilisent une méthode d'interprétation abstraite sur le contenu des registres du processeur pour identifier les chemins d'exécution infaisables. Ermedahl et Gustafsson [10] utilisent une méthode d'interprétation abstraite sur le code source pour obtenir l'ensemble des informations de flot de contrôle.

3.2 Calcul du WCET

Une fois les informations de flot de contrôle obtenues, il faut, à partir de leur représentation, identifier le chemin d'exécution le plus long. La méthode utilisée pour la recherche du plus long chemin permet de distinguer les différentes classes de méthodes de calcul des WCET. Pour simplifier, on suppose ici que le temps d'exécution de chaque instruction (et par conséquent de chaque bloc de base) est connu et constant, quel que soit son contexte d'exécution. Cette hypothèse simplificatrice, qui est à la base des premiers travaux du domaine, nous permet de ne pas nous préoccuper pour l'instant du niveau bas de l'analyse statique de WCET. Cette hypothèse sera levée au paragraphe 4.

3.2.1 Techniques utilisant l'algorithmique des graphes (*Path-based*)

La connaissance des WCET individuels des blocs de base permet d'associer des WCET aux nœuds du graphe de flot de contrôle. On obtient alors un graphe valué avec un seul point d'entrée et un seul point de sortie. On peut chercher le pire chemin d'exécution dans le graphe de flot de contrôle [18, 8] en utilisant les algorithmes traditionnels de l'algorithmique des graphes. Puis on vérifie que le chemin trouvé est un chemin d'exécution possible, et si ce n'est pas le cas, on l'exclut du graphe et on recommence la recherche. Les informations de nombre maximum d'itérations limitent le nombre d'occurrences d'un bloc de base ou d'un arc dans un chemin d'exécution.

3.2.2 Techniques *IPET*

Cette classe de méthodes, dite *d'énumération implicite des chemins* (ou *IPET - Implicit Path Enumeration Technique*) est utilisée dans de nombreux travaux d'analyse statique de WCET [19, 20, 21, 22]. Cette approche ne s'appuie que sur la représentation du programme sous forme de graphe de flot de contrôle, qu'elle transforme en un ensemble de contraintes devant être respectées. Un premier jeu de contraintes décrit la structure du graphe, et un deuxième permet de prendre en compte les informations supplémentaires sur le flot de contrôle (§ 3.1.1).

La figure 4 montre le système de contraintes généré pour l'exemple de la figure 2. Chaque nœud du graphe de flot de contrôle est valué par n_i, son nombre d'occurrences dans le pire chemin d'exécution. À chaque nœud du graphe, la somme des nombres d'occurrences des nœuds prédécesseurs doit être égale à celle des nombres d'occurrences des nœuds successeurs. On obtient ainsi un premier système de contraintes qui décrit la structure du graphe (contraintes (a) sur la figure). Par exemple, la contrainte $n_1 = n_2 + n_3$ indique que la somme des nombres d'exécution des blocs de base BB_2 et BB_3 (branches de la structure conditionnelle de la fonction *impaire*) est égale au nombre d'exécutions du bloc de base BB_1 (test de la structure conditionnelle). Dans le deuxième système de contraintes (contraintes (b) sur la figure), on trouve, par exemple, les contraintes sur les nombres maximum d'itérations des boucles.

Étant donnés ces deux systèmes de contraintes, on cherche à maximiser l'expression du WCET :

$$WCET = \sum_i n_i \times w_i$$

où w_i est le WCET du bloc de base BB_i. Les valeurs des w_i sont fournies par l'analyse de bas niveau décrite au paragraphe 4.

Les techniques de calcul IPET ne font appel qu'au graphe de flot de contrôle du programme, autorisant ainsi la présence de code ayant fait l'objet d'optimisations lors de la compilation. De plus, elles permettent d'ajouter d'autres contraintes

	Formules pour le calcul du WCET : $W(S)$
$S = S_1; ...; S_n$	$WCET(S) = WCET(S_i) + ... + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test)$ $+\max(WCET(S_1), WCET(S_2))$
$S =$ loop(tst) S_1	$WCET(S) = maxiter \times (WCET(Test) + WCET(S_1))$ $+WCET(Test)$ où $maxiter$ est le nombre maximum d'itérations.
$S = f(exp_1, \cdots, exp_n)$	$W(S) = W(exp_1) + \cdots + W(exp_n) + W(f())$
$S =$ bloc de base BB_x	$WCET(S) =$ WCET du bloc de base BB_x

TAB. **1 –.** *Formules de calcul du WCET [9]*

que celles liées aux boucles pour, par exemple, prendre en compte l'exclusion mutuelle de deux blocs de base (par exemple, $n_\alpha + n_\beta \leq 1$ implique l'exclusion mutuelle entre les nœuds α et β). Cette possibilité est exploitée par exemple dans [20].

La maximisation de l'expression du WCET ci-dessus fournit les valeurs des n_i (et, par suite, le WCET). Les méthodes de résolution utilisées sont similaires à celles utilisées pour résoudre les problèmes de programmation linéaire (Integer Linear Programming) ou de satisfaction de contraintes. Le temps d'analyse est fonction de la complexité du système et peut donc être important [23]. Par ailleurs, hormis les valeurs des n_i et du WCET, ce type de méthode n'identifie pas explicitement le pire chemin d'exécution, ce qui peut rendre difficile l'utilisation de ce type de technique à des fins d'optimisation de code.

3.2.3 Techniques basées sur l'arbre syntaxique (*Tree-based*)

Cette classe de méthodes, proposée initialement par Puschner et Koza dans [9], s'appuie sur la représentation du programme en arbre syntaxique pour calculer récursivement son WCET. Un ensemble de formules (*cf.* tableau 1) permet d'associer à chaque structure syntaxique du langage source (un nœud de l'arbre syntaxique) un WCET, et ce en fonction des sous-arbres qui la composent (les fils du nœud) jusqu'à arriver aux feuilles de l'arbre que sont les blocs de base dont on suppose les WCET connus. On effectue donc un parcours de bas en haut (*bottom-up*) de l'arbre en partant des feuilles porteuses de l'information de WCET, pour obtenir le WCET de la racine.

À chaque nœud de l'arbre où un choix est possible, on choisit le chemin qui maximise le temps d'exécution. Par exemple, le WCET d'une séquence est simplement la somme des WCET des structures qui la composent et le WCET d'une conditionnelle implique l'utilisation de l'opérateur *max* pour choisir la branche conditionnelle dont le temps d'exécution est le plus important.

Les informations concernant les boucles, définies dans le paragraphe 3.1.1, sont prises en compte par des formules associées aux structures de boucle. L'énumération de tous les chemins d'exécution possibles est réalisée par l'application récursive des équations le long de toute la structure arborescente du programme.

On obtient ainsi un arbre temporel (*timing tree*) [24] qui contient les WCET calculés aux différents nœuds de l'arbre. Il est ainsi aisé d'utiliser ce type de méthode pour l'optimisation de code.

4. Analyse de bas niveau

Une technique d'analyse statique de WCET "naïve" repose sur deux hypothèses: (i) elle suppose que le WCET d'une séquence de deux blocs de base est égal à la somme des WCET des blocs de base pris séparément, (ii) elle suppose que le WCET d'un bloc de base est constant quelque soit son contexte d'exécution (*i.e.* son WCET ne dépend pas de l'enchaînement de blocs de base le précédant dans un chemin d'exécution). Ces hypothèses simplificatrices ne sont vérifiées que si on ignore l'effet de certains éléments de l'architecture matérielle qui permettent d'améliorer les performances moyennes du système (par exemple les caches et pipelines). Ainsi, le pipeline, en introduisant du parallélisme dans le traitement d'une suite d'instructions, remet en cause la premiere hypothèse. De même, le cache d'instructions introduit une variation du temps de traitement d'une instruction en fonction du contexte, ce qui contrarie la deuxième hypothèse.

La modélisation du comportement de ces éléments d'architecture n'est pas triviale, et leur prise en compte introduit une dépendance au contexte. Cependant, l'intégration de ces éléments dans l'estimation du WCET permet d'améliorer considérablement sa précision tout en garantissant la sûreté des estimations. C'est pourquoi de nombreux travaux concernent la prise en compte de ces éléments, principalement les caches [25, 26, 27, 28] et les pipelines [29, 30].

On peut classer les effets de l'architecture matérielle sur le temps d'exécution en deux catégories: les effets *locaux* et les effets *globaux*. Un élément d'architecture a un effet local lorsque, par l'entremise de cet élément, le comportement temporel d'une instruction ne peut affecter le comportement d'une autre instruction que si celle-ci est "proche" dans le flot d'instruction. C'est typiquement le cas du pipeline (simple) pour lequel le comportement tem-

porel d'une instruction n'affecte que les quelques instructions suivantes. On dit d'un élément d'architecture qu'il a un effet global s'il permet à une instruction d'affecter le comportement d'autres instructions quelles que soient leurs distances dans le flot d'exécution. C'est par exemple le cas du cache d'instructions pour lequel le chargement d'une instruction peut causer le remplacement d'une autre instruction et ainsi affecter le temps d'exécution de cette dernière, même si celle-ci n'est exécutée que bien plus tard.

Dans les paragraphes qui suivent, nous donnons quelques exemples de prise en compte des effets de l'architecture matérielle sur l'obtention des WCET, que nous classons selon les deux catégories précédentes.

4.1 Analyse de bas niveau globale : le cas des caches d'instructions

Parmi les éléments d'architecture les plus souvent pris en compte pour l'analyse de WCET, les caches d'instructions [28, 31, 25] ont un effet global. D'autres éléments ayant un effet global, comparativement moins étudiés que les caches d'instructions sont les caches de données [32] et les prédicteurs de branchement [33]. Nous nous limitons ici pour des raisons de place à la description d'une des méthodes de prise en compte des caches d'instructions. D'autres méthodes d'analyse de bas niveau globale pourront être trouvées dans [1].

L'usage d'une mémoire cache introduit de l'indéterminisme dans l'architecture. Il faut pouvoir estimer de façon *sûre* (et pas seulement de manière probabiliste) le résultat (succès/échec) des accès au cache et pour cela connaître statiquement le pire comportement des accès mémoire vis à vis du cache lors de l'exécution d'un programme. Ainsi, seul les accès mémoire pour lesquels on est certain que l'accès au cache sera un succès sont classés comme tel (s'il y a un doute, on considère l'estimation la plus pessimiste : échec). Les méthodes ayant pour but d'intégrer le comportement du cache d'instructions dans le calcul du WCET ont pour objectif commun d'effectuer une *classification* de toutes les instructions d'un programme en fonction de leur pire comportement par rapport au cache.

La méthode décrite dans [28, 31], nommée par ses auteurs *simulation statique de cache*, consiste à envisager statiquement et en même temps tous les chemins d'exécution possibles du graphe de flot de contrôle, de manière à calculer une représentation de tous les contenus possibles du cache à différents moments de l'exécution.

Cette méthode distingue quatre catégories d'accès aux instructions : *always hit*, *always miss*, *first miss* ou *conflict*, selon que l'on peut ou non garantir statiquement leur présence dans le cache d'instructions au moment de leur exécution. Les accès en mémoire aux instructions classés *always hit* sont toujours des succès, et les accès aux instructions classés *always miss* sont considérés comme étant toujours des échecs. La catégorie *first miss* indique une instruction qui cause un échec uniquement la première fois qu'elle est exécutée (dans une boucle par exemple). Enfin, la dernière catégorie, *conflict*, indique qu'un accès à une instruction est considéré comme un échec car on ne peut obtenir d'estimation sûre de son comportement (l'échec constituant le cas pire du point de vue du temps d'exécution).

Cette classification des instructions par rapport au cache est effectuée en deux étapes. La première étape calcule, pour chaque bloc de base, tous les contenus possibles du cache, avant et après exécution du bloc. Ces contenus sont appelés *états abstraits de cache*, car ils ne représentent pas un contenu du cache en suivant un chemin d'exécution particulier, mais tous les contenus possibles en considérant tous les chemins simultanément. Les états abstraits de cache sont obtenus par un calcul de point fixe sur le graphe de flot de contrôle. Cette première étape permet, pour chaque bloc de base, d'identifier les instructions en conflit pour chaque ligne de cache. La deuxième étape effectue la classification à partir des états abstraits de cache.

La classification peut alors être utilisée pour calculer les WCET de chacun des blocs de base, à leur tour utilisés par la méthode de calcul.

4.2 Analyse de bas niveau locale : le cas des pipelines

Le pipeline est une technique dans laquelle plusieurs instructions se recouvrent au cours de leur exécution. C'est la technique fondamentale utilisée pour réaliser des processeurs rapides. Afin d'expliciter brièvement le principe du pipeline, on rappelle que le cycle d'exécution d'une instruction se décompose en plusieurs étapes, par exemple : lecture d'instruction, décodage d'instruction, exécution, accès mémoire, écriture du résultat. Le but du pipeline est d'introduire du parallélisme entre les traitements de différentes instructions. Pour ce faire, il comporte plusieurs étages qui lui permettent de traiter en parallèle les différentes étapes des instructions. Les exécutions des instructions se chevauchent et le temps d'exécution de deux instructions dans le pipeline n'est pas la somme de leurs temps d'exécution unitaires, car le traitement d'une instruction peut débuter avant la fin de celle de l'instruction précédente. Ces différentes étapes sont illustrées sur la figure 5.

On voit bien d'après cette description que le pipeline est un élément d'architecture dont les effets sont locaux. Si on considère une séquence d'instructions exécutées dans un pipeline, l'effet d'une instruction porte sur les quelques instructions suivantes et devient nul au bout d'un certain temps. Nous examinons ici les méthodes de prise en compte de l'exécution pipelinée dans le cas mono-pipeline et pour les processeurs qui exécutent les instructions dans l'ordre de leur apparition dans le programme.

Les effets du pipeline sur le calcul du WCET se retrouvent à deux niveaux : (i) intra blocs de base pour la prise en compte du chevauchement entre instructions et des aléas de données et (ii) inter blocs de base pour la prise en compte des aléas de contrôle et du chevauchement entre blocs.

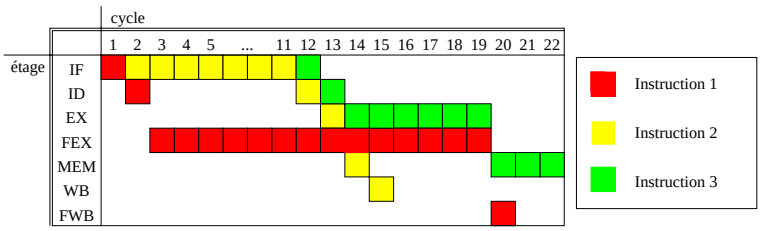


FIG. **5 –.** *Principe du pipeline*

Au sein d'un bloc de base, aucun aléa de contrôle ne peut avoir lieu pendant l'exécution. Toutefois, la présence de dépendances de donnée et d'aléas de structure est possible. Pour prendre en considération ces deux aspects, le calcul du WCET des blocs de base présenté dans [34] repose sur la représentation de l'occupation du pipeline par des tables de réservation pendant l'exécution de ces blocs. L'état d'occupation des étages du pipeline est simulé par remplissage des tables de réservation. Ainsi sont prise en compte à la fois du recouvrement de l'exécution d'instructions consécutives du même bloc de base et la présence de dépendances de données et d'aléas de contrôle.

Le pipeline a également un impact entre blocs de base exécutés successivement, et l'on doit, comme on l'a fait pour le calcul du temps d'exécution d'un bloc de base, prévoir une concaténation avec recouvrement des descripteurs de pipelines. Notons toutefois qu'il n'est pas utile de conserver l'intégralité des descripteurs de pipelines des blocs de base : le début et la fin des descripteurs suffisent. Notons également qu'il est nécessaire de prévoir des aménagements à cette méthode dans les architectures dotées d'un prédicteur de branchement, afin de prendre en compte les branchements mals prédits.

5. Tendances actuelles

Dans cet article, nous avons dressé un panorama des méthodes de calcul du temps d'exécution pire cas. La plupart des avancées dans ce domaine sont le fruit de recherches académiques et n'ont peut-être pas tout à fait encore trouvé un écho significatif dans le milieu industriel. Toutefois, les discussions que nous pouvons avoir avec les industriels montrent bien que les processeurs qu'ils utilisent aujourd'hui les amènent à reconsidérer leurs méthodes de calcul de temps d'exécution, et que l'intérêt qu'ils portent aux techniques d'obtention des WCET est croissant.

5.1 Limites actuelles de l'analyse statique de WCET

Il y a encore quelques années, les processeurs utilisés dans des systèmes temps-réel étaient relativement simples et ne posaient pas de gros problèmes pour le calcul de WCET parce que le temps d'exécution d'une portion de code était indépendant du contexte d'exécution, et pouvait donc être mesuré de manière indépendante. Or, on constate une évolution fulgurante de l'architecture des processeurs utilisés dans les systèmes temps-réel : d'une part, les processeurs spécialisés ont tendance à intégrer les mécanismes présents dans les processeurs haute-performance (avec quelques années de retard), et d'autre part, les besoins en performances étant croissants, les concepteurs sont de plus en plus tentés d'utiliser des processeurs haute-performance non spécifiques. Aussi, les méthodes "artisanales" de calcul de WCET ne sont plus suffisantes maintenant et les techniques d'analyse doivent permettre de prendre en compte les spécificités des processeurs les plus récents.

Du point de vue de la modélisation du processeur, les techniques dynamiques sont capables de prendre en compte n'importe quelle architecture. Cependant, nous avons vu que ces techniques nécessitent d'être capable de définir des jeux d'entrées couvrant tous les chemins possibles, ce qui est loin d'être toujours le cas, sans quoi les résultats obtenus ne peuvent pas être considérés comme sûrs. Les techniques utilisant des jeux de tests symboliques quant à elles se heurtent à des problèmes de temps d'estimation des WCET pour des programmes de taille importante.

Dans le cadre d'une analyse statique, on n'est pas capable, à ce jour, de modéliser bon nombre de mécanismes présents dans les processeurs haute performance. En effet, la plupart de ces mécanismes ont un comportement hautement dynamique, qui ne peut être prédit exactement de manière purement statique. Toutefois, on peut distinguer différents degrés dans le caractère dynamique des processeurs : certains comportements dynamiques peuvent être appréhendés de manière approchée par analyse statique, d'autres demeurent hors de portée des techniques actuelles.

La principale difficulté de l'analyse statique de mécanismes dont le comportement est lié à l'historique d'exécution est la nécessité de considérer les chemins d'exécution dans leur globalité pour connaître cet historique. Les méthodes statiques n'exécutant pas les chemins un à un, elles doivent synthétiser, pour un point donné du programme, un historique *abstrait* représentant tous les historiques possibles (voir par exemple § 4.1). Toutefois, il n'est pas applicable pour tous les mécanismes. Pour d'autres mécanismes, une analyse basée sur une synthèse des historiques n'est pas appropriée. C'est par exemple le cas pour les pipelines. Nous avons vu (§ 4.2) que les pipelines pouvaient être analysés sur des portions de chemins pour des pipelines scalaires (une seule instruction à la fois dans chaque étage) et une exécution des instructions dans l'ordre. Pour des pipelines plus complexes (superscalaires, avec une exécution des instructions éventuellement dans le désordre) les méthodes

statiques actuelles sont encore incapables de modéliser correctement la plupart des processeurs haute-performance [35].

Enfin, le principe d'exécution spéculative, c'est-à-dire la possibilité d'exécuter les instructions sur le mauvais chemin en attendant la résolution d'un branchement prédit de manière incorrecte, constitue une autre source de complication. En effet, la plupart des méthodes statiques actuels considère uniquement les chemins corrects. Ainsi, les instructions exécutées sur des chemins incorrects ne peuvent pas être prises en compte. Or, il a été montré que l'exécution des chemin incorrects pouvait avoir un certain nombre d'effets sur l'exécution des chemins corrects : altération du contenu des tables de prédiction de branchement, modification du contenu des caches, ces effets pouvant aussi bien augmenter que diminuer le WCET. Aussi, l'exécution spéculative n'est pas compatible avec un calcul de WCET fiable par les méthodes d'analyse statiques actuelles.

Un autre point à souligner est que la plupart des études sur le WCET se focalisent sur un mécanisme matériel et très peu portent sur les interactions entre mécanismes. Or, Lundqvist [36] a montré que, dans le cas d'un processeur à exécution non ordonnée, un échec dans le cache concernant une instruction d'un bloc de base pouvait conduire à un temps d'exécution global plus court de ce bloc de base. Ainsi, ce qui jusqu'à présent semblait acquis comme étant le cas pire (échec dans le cache) se révèle ne pas l'être forcément du fait des répercussions d'un mécanisme sur l'autre. Cette incertitude sur le cas pire constitue une difficulté supplémentaire.

En marge de ces considérations, la liste des mécanismes dont la prise en compte par des méthodes statiques n'a pas encore été examinée à ce jour est longue. On peut citer, par exemple, le cache de traces présent dès les premières versions du Pentium 4 ou encore l'exécution multiflot introduite dans sa dernière version, mais aussi le mécanisme de mémoire virtuelle intégré dans tous les processeurs modernes et dans certains processeurs embarqués. D'autres mécanismes ont été étudiés, mais de manière incomplète : caches et prédicteurs de branchement, pour lesquels tous les algorithmes existants n'ont pas été examinés. Il est difficile de dire, sans avoir mené d'étude précise, si ces mécanismes peuvent être facilement intégrés dans les méthodes de calcul de WCET actuelles. Mais il est probable qu'ils feront l'objet de travaux dans les années à venir.

5.2 Perspectives

L'obtention des WCET par analyse statique trouve ses limites pour les architectures complexes (pessimisme excessif et/ou non existence de méthodes d'analyse adaptées). Plusieurs voies nous semblent intéressantes à explorer pour contourner ces limites :

- *Utiliser du matériel au comportement temporel prévisible* : éviter les éléments architecturaux que l'on ne sait pas prendre en compte correctement ou, quand c'est possible, configurer le matériel pour rendre son comportement prévisible. Par exemple, le mécanisme de verrouillage de cache [37] (de données d'instructions, ou de cache de traduction d'adresses) permet de rendre la hiérarchie de mémoire déterministe. Il est également souvent possible de désactiver certains mécanismes, comme la mémoire virtuelle ou la prédiction de branchement. En outre, certains processeurs permettent de configurer des mécanismes spéculatifs dynamiques (par exemple, les prédicteurs de branchement) en mode statique [38].
  Un argument en faveur de cette approche est que le WCET calculé pour une architecture simple n'est pas forcément très éloigné de celui que l'on évaluerait de manière très pessimiste pour une architecture avancée que l'on ne saurait pas modéliser finement. On peut également penser à concevoir des processeurs adaptés aux applications temps-réel et au calcul de WCET. Ils pourraient intégrer des mécanismes qui assurent une certaine prévisibilité (du point de vue temporel) des comportements dynamiques et spéculatifs.
- *Définir des compromis acceptables entre sûreté des estimations et précision*. D'un côté, les estimations obtenues par les méthodes statiques sont sûres, mais pessimistes ; de l'autre, celles obtenues par les méthodes dynamiques sont plus précises, mais pas toujours très fiables, sauf si l'on est capable d'énumérer exhaustivement les entrées du programme. Il est probablement souhaitable de mettre en place des méthodes intermédiaires. C'est ce qui est proposé dans [39] : une méthode mixte alliant des tests (mesures sur quelques chemins) pour l'estimation des temps d'exécution des blocs de base et un calcul basée sur la structure du code pour aggréger les WCET des blocs de base et ainsi obtenir le WCET du programme. Le WCET obtenu est moins pessimiste que celui qui serait calculé de manière statique, mais il n'est connu que de manière probabiliste.
  On peut actuellement observer un regain d'intérêt envers les méthodes dynamiques, qui possèdent l'avantage indéniable de ne pas nécessiter de modèle temporel du processeur. Les objectifs des travaux en la matière visent à limiter la combinatoire de l'énumération des jeux d'entrée, en utilisant des méthodes issues du monde du test logiciel [40] ou en n'appliquant les méthodes d'énumération des jeux d'entrée uniquement sur des fragments de programme [41].

Parallèlement au développement de stratégies visant à améliorer la capacité à calculer un WCET à la fois sûr et réaliste (c'est-à-dire pas trop surestimé), il est probablement souhaitable de lier les outils de calcul statique de WCET aux compilateurs :

- l'analyse de flot nécessaire au calcul de WCET pourrait être plus efficace puisqu'elle pourrait bénéficier de la connaissance que le compilateur a du code source

et des transformations réalisées dans le cadre des optimisations.

– le compilateur pourrait faire appel à l'outil d'analyse de WCET pour évaluer la pertinence des optimisations : d'une part, la connaissance du chemin le plus long lui permettrait de savoir sur quelle portion de code les efforts doivent être portés pour réduire le WCET ; d'autre part, l'impact d'une optimisation sur le WCET pourrait être calculé.

6. Pour aller plus loin

Deux numéros de la revue *Journal of Real Time Systems* (volumes 17 et 18, publiés respectivement en novembre 1999 et mai 2000) sont entièrement consacrés aux méthodes d'obtention des WCET. Par ailleurs, un *Workshop* est organisé annuellement sur ce thème, en conjonction avec la *Euromicro Conference on Real Time Systems*. Ce *Workshop*, dont les actes sont disponibles en ligne (*http://www.ecrts.org/wcet/*) permet d'observer les tendances les plus récentes en terme d'obtention de WCET.

Références

[1] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques*, 22(5):651–677, 2003.

[2] G. C. Buttazzo, editor. *Hard real-time computing systems - predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[3] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 2001.

[4] N. Tracey, J. Clark, and K. Mander. The way forward for unifying test case generation: The optimisation-based approach. In *IFIP Workshop on Dependable Computing and Its Applications*, pages 73–81, janvier 1998.

[5] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[6] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS95)*, pages 298–307, Pisa, Italy, December 1995.

[7] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[8] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[9] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[10] A. Ermedahl and J. Gustafsson. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2):61–74, 1998.

[11] D. Macos and F. Mueller. Integrating gnat/gcc into a timing analysis environment. In *Work-in-Progress of the 10th Euromicro Conference on Real-Time Systems*, pages 15–18, June 1998.

[12] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, December 1996.

[13] R. Chapman, A. Burns, and A.J. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[14] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[15] L. Ko, D. B. Whalley, and M. G. Harmon. Supporting user-friendly analysis of timing constraints. *ACM SIGPLAN Notices*, 30(11):99–107, November 1995.

[16] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3):129–156, May 2000.

[17] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for real-life processor. In *Proceedings of the first international workshop on embedded software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Sciences*, pages 469–485, Tahoe City, CA, USA, October 2001.

[18] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[19] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler technique to cache behavior prediction. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 37–46, June 1997.

[20] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 30(11):88–98, November 1995.

[21] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems (LCTRTS'97)*, June 1997.

[22] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat, Institut fur Technische Informatik, Wien, April 1995.

[23] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, December 1996.

[24] P. Puschner. A tool for high-level language analysis of worst-case execution times. In *Proceedings of the 10th Euromicro Conference on Real-Time Systems*, pages 130–137, Berlin, Germany, June 1998.

[25] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66, September 1996.

[26] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *International Conference on Computer Aided Design*, pages 380–387, Los Alamitos, Ca., USA, November 1995.

[27] S.-S Lim, S. L. Min, M. Lee, C. Y. Park, H. Shin, and C.-S. Kim. An accurate instruction cache analysis technique for real-time systems. *IEEE Workshop on Architectures for Real-Time Applications*, April 1994.

[28] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS94)*, pages 172–181, December 1994.

[29] S. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical Report TR94-072, University of North Carolina, Chapel Hill, April 1995.

[30] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

[31] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[32] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 255–262, 1999.

[33] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[34] B.-D. Rhee, S.-S. Lim, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. Issues of advanced architectural features in the design of a timing tool. In *Proceedings of the 11th Workshop on Real-Time Operanting Systems and Software*, pages 59–62, May 1994.

[35] J. Engblom. *Processor Pipelines and Static WCET Analysis*. PhD thesis, Uppsala University, April 2002.

[36] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[37] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 114–123, Austin, Texas, December 2002.

[38] François Bodin and Isabelle Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[39] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, Austin, Texas, December 2002.

[40] N. Williams. Wcet measurement using modified path testing. In *Proceedings of the 5th International Workshop on worst-case execution time analysis, in conjunction with the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[41] J.F. Deverge and I. Puaut. Safe meaurement-based wcet estimation. In *Proceedings of the 5th International Workshop on worst-case execution time analysis, in conjunction with the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

Conception conjointe commande/ordonnancement et ordonnancement régulé

Daniel Simon
Inria Rhône-Alpes – Projet POP ART
655 avenue de l'Europe Montbonnot
38334 St Ismier Cedex
Daniel.Simon@inrialpes.fr

Abstract

Les performances de commande d'un système en boucle fermée dépendent aussi bien de l'algorithme utilisé que des paramètres d'exécution des programmes, telles que périodes, latences et gigue. Les tâches de commande sont soumises à des incertitudes temporelles dues à l'exécution des programmes sur la ressource de calcul. Conjointement à l'algorithme de commande les paramètres d'ordonnancement d'un programme de commande doivent être adaptés ou optimisés en vue d'améliorer les performances d'un contrôleur, en particulier sa robustesse temporelle. D'autre part les techniques de commande en boucle fermée peuvent être appliquées à l'ordonnanceur temps-réel lui même, le rendant ainsi adaptatif et robuste face aux incertitudes temporelles d'exécution. Cette démarche de conception conjointe et d'ordonnancement régulé est illustrée par quelques exemples.

1 Introduction : Commande et Rétroaction

Le but d'un système de commande est de contrôler un processus pour l'amener dans un état conforme aux désirs de l'utilisateur. Il est depuis longtemps apparu que la réalisation d'une commande en boucle fermée, où la valeur des sorties du processus rétroagit sur le signal de commande, permet d'améliorer les performances du système, exprimées par exemple en terme de précision de suivi de consignes ou d'insensibilité aux perturbations [15]. Le contrôleur réalisant la fonction de commande est le plus souvent réalisé sur un calculateur numérique suivant le schéma représenté par la figure 1. Les composants principaux du système sont :

– Le processus commandé : c'est le plus souvent un processus physique (machine électromécanique, processus chimique…) mais ce peut être aussi une entité informatique (serveur multimédia). Il est caractérisé par une dynamique d'entrée/sortie :

$$\begin{cases} \dot{X} = f(X,U,t) & \text{équation d'état} \\ Y = g(X,U,t) & \text{équation de mesure} \end{cases}$$

où la sortie observée Y dépend de l'état interne X du système, de sa commande U et du temps t.

– Les capteurs permettent de mesurer les sorties (continues) du système. Elles sont échantillonnées (au moyen d'une horloge de période h)pour être transmises au calculateur numérique ;

– Les actionneurs reçoivent des commandes et agissent sur le processus. Les commandes calculées numériquement doivent être bloquées (le plus souvent en pratique par un bloqueur d'ordre zéro) pour pouvoir agir entre les instants d'échantillonnage ;

– Le régulateur K calcule les commandes en fonction de l'écart e entre la mesure Y et la consigne Y_d au moyen d'une fonction plus ou moins complexe et coûteuse. Il peut aussi inclure des fonctions de filtrage, de reconstruction d'état et de pré-compensation.

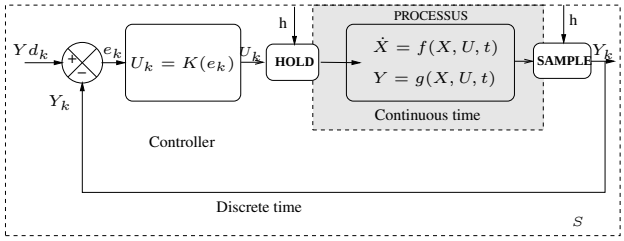


FIG. **1. Commande en boucle fermée**

Par rapport à une commande en boucle ouverte (sans rétroaction) les bénéfices attendus d'une commande en boucle fermée bien conçue sont multiples et permettent potentiellement d'améliorer les performances de commande sur plusieurs points :

– Stabilité (au sens entrée bornée/sortie bornée) : capacité à stabiliser un processus naturellement instable ;

– Augmentation de la précision en régulation et/ou en poursuite ;

– Accélération du temps de réponse sans sollicitations excessives des actionneurs ;
– Rejet de perturbations externes, mesurées ou non ;
– Robustesse aux incertitudes de modèle, garantissant un certain niveau de stabilité et de performances ;

Toutes ces performances ne peuvent être simultanément et arbitrairement améliorées, le concepteur du contrôleur doit gérer un compromis entre stabilité/précision/saturations/sensibilité/robustesse. En particulier une grande robustesse aux incertitudes de modèle du processus commandé se paye par un niveau de performance modéré en terme de temps de réponse et de bande passante (théorème du petit gain e.g.[43]).

Enfin, au delà des aspects purement algorithmiques, la réalisation d'une boucle de commande sur un calculateur numérique va perturber plus ou moins gravement ses performances par l'effet de l'échantillonnage et de retards induits.

Pour bénéficier des avantages de la commande fermée le contrôleur doit être correctement conçu, réalisé et paramétré. En revanche, une mauvaise conception peut entraîner une déstabilisation du système et un risque de divergence encore plus rapide qu'en boucle ouverte. Les divers aspects concernant l'algorithmique de commande et l'implémentation du contrôleur doivent être idéalement pris en compte simultanément pour tirer tous les bénéfices de l'approche boucle fermée.

2 Implémentation des systèmes de commande

Les systèmes de commande en boucle fermée sont pour la plupart périodiques, les entrées (lecture de capteurs), les calculs de commandes et les sorties (envoi de commandes aux actionneurs) sont effectués à une période unique et constante. Cependant, bien que la théorie classique de la commande des systèmes numérique repose sur l'utilisation d'un échantillonnage à période unique fixe, il a été mis en évidence, e.g. [3], que la réalisation d'un contrôleur sous forme multi-tâches et multi-cadences permet d'en améliorer les performances, même dans le cas de systèmes linéaires simples [4]. En effet, certaines parties d'un algorithme de commande (par exemple la mise à jour de paramètres) peuvent être moins urgentes à calculer que l'élaboration de la commande des modes rapides du système : leur calcul peut être sans dommage décalé dans le temps ou être exécuté à une cadence plus lente. En fait un système complexe est constitués de sous-ensembles aux dynamiques différentes devant être exécutées, puis coordonnées, en fonction de leurs caractéristiques temporelles respectives [44]. Le système de contrôle/commande doit donc gérer en parallèle (et en temps-réel) un certains nombre de contrôleurs élémentaires exécutés à leur cadence propre dans une hiérarchie de niveaux plus ou moins couplés.

Les applications de contrôle/commande, que l'on trouve par exemple en robotique, nécessitent d´exécuter des activités informatiques ayant des caractéristiques temporelles différentes ; on trouve par exemple :

– des lois de commande périodiques, où le respect de contraintes de cadence et de latence conditionne les performances de la commande ;
– des lois de commande périodiques multi-cadences, où on peut affecter des caractéristiques temporelles (période, priorité...) différentes à certaines parties de l'algorithme pour en augmenter l'efficacité [40] ;
– des activités, répétitives ou non, dont la durée d'exécution dépend de l'environnement et n'est pas connue a priori (planification, traitement d'images...) ;
– des activités sporadiques, représentant par exemple les changements d'état et de configuration du système, les traitements d'exceptions ou les réactions aux pannes.

Ces diverses activités partagent un support d'exécution de capacité bornée en devant respecter des contraintes temps-réel spécifiées sous la forme d'échéances temporelles à respecter. Le respect de ces échéances peut être strict (tout dépassement est interdit) ou lâche (les commandes périodiques peuvent tolérer sans dommage des variations autour de valeurs nominales de périodes et de latences). On peut aussi graduer une contrainte stricte et définir des systèmes ¨weakly hard¨, où la contrainte dure de respect d'échéance est remplacée une distribution précisément définie de dépassements autorisés, exprimée en terme du nombre d'échéances respectées ou dépassées sur une fenêtre temporelle donnée [5]. Le modèle d'erreurs temporelles doit être complété par la définition du traitement d'exception associé devant idéalement permettre un passage graduel en fonctionnement dégradé.

3 Modèle classique d'ordonnancement temps-réel

L'exécution de ces systèmes est souvent gérée par un système d'exploitation utilisant un ordonnanceur à priorités fixes et préemption. La variété des caractéristiques temporelles et l'incertitude liée aux valeurs connues uniquement a l'exécution de ces activités font qu'il est difficile et inefficace de spécifier un ordonnancement statique (e.g. une affectation statique de périodes), utilisant des majorants et conduisant a une sous-utilisation chronique du support d'exécution. Dans ce cas les dépassements d'échéance peuvent également conduire à une avalanche de fautes temporelles et à un écroulement du système. D'autre part, dans la littérature classique concernant l'ordonnancement temps-réel, les algorithmes (par exemple la politique d'affectation des priorités) sont étudiés séparément du contexte applicatif [13]. Cette démarche peut conduire a préconiser un ordonnancement ¨optimal¨ du point de vue par exemple de l'utilisation de la ressource de calcul mais inefficace du point

de vue du contrôle de l'application qui est tout de même le point de départ [44].

Rappelons que d'une façon classique le système informatique est modélisé par un ensemble $\{\tau_1, \tau_2, ..., \tau_n\}$ de tâches, chacune d'entre elle étant caractérisée par ses paramètres temporels [25] :

- la date de première activation r_i
- la pire durée d'exécution (WCET) C_i
- le délai critique relatif (deadline) D_i
- la période d'activation P_i

Sous les hypothèses restrictives suivantes :

- toutes les tâches sont périodiques et s'exécutent sur un mono-processeur ;
- elles ont le même instant critique (tous les r_i sont égaux) ;
- il n'y a pas de relation de dépendance entre les tâches ;
- les tâches sont à échéance sur requête ($P_i = D_i$, pour tout i)

il est montré que la politique Rate Monotonic (les priorités fixes sont ordonnées dans l'ordre inverse des périodes) est optimale, i.e. tout ensemble de tâches ordonnançable de cette classe l'est par R.M. (et EDF est optimal pour des priorités dynamiques). De nombreuses extensions ont par la suite été apportées au modèle original pour, par exemple, prendre en compte les dépendances et l'exclusion mutuelle entre tâches ou encore l'existence de processus apériodiques [38].

3.1 Limites de l'approche

Connaissance des paramètres temporels Ces approches supposent que l'on connaisse au départ la valeur des paramètres temporels des tâches. Si toutes les tâches ne sont pas périodiques, le problème peut être plus ou moins bien contourné par une ¨mise en réserve¨ de temps CPU (e.g. le serveur sporadique), ce qui suppose tout de même de connaître une borne inférieure du temps séparant deux activations successives. On peut difficilement intégrer des notions d'urgence ou d'importance de ces tâches.

La détermination de la durée d'exécution des tâches est un problème difficile. Cette durée dépend évidement du langage hôte (et des optimisations éventuelles du compilateur) et de la machine cible. Pour une machine et un compilateur donné, cette durée dépend également du contexte, et est de plus en plus difficile à déterminer avec les processeurs modernes utilisant des caches et/ou des pipe-lines. Les branchements conditionnels dans le code génèrent également des durées d'exécution variables (potentiellement une durée par branche).

L'incertitude sur la durée d'exécution des tâches peut également provenir de l'algorithme lui même. Citons en particulier les processus de vision où la durée de traitement dépend de la richesse et de la complexité de la scène observée, et les algorithmes de planification de trajectoires... Les processus itératifs, où la précision du résultat dépend du nombre d'itérations effectuées, fournissent un exemple de processus où la valeur d'un critère de qualité de services peut être calculée en fonction de la précision du résultat, et donc plus ou moins explicitement en fonction de la durée de calcul effectuée. C'est en particulier le cas des contrôleurs prédictifs (model predictive control MPC), où une optimisation quadratique doit être effectuée à chaque pas [21].

Enfin, remarquons que même une connaissance de la durée d'exécution au pire cas ne garantit pas forcément l'obtention d'un ordonnancement fiable. Par exemple, [23] analyse le fonctionnement d'une installation industrielle où l'ordonnancement se déroule conformément au cahier des charges lorsque toutes les tâches sont à leur WCET et qui se bloque sur une inversion de priorité non protégée quand une tâche d'acquisition termine son exécution plus tôt que prévu...

Politique d'affectation des priorités Les politiques d'affectation des priorités étudiées dans la littérature (RM, DM, EDF...) présentent un certain caractère d'optimalité, e.g. utilisation maximale du CPU, donc du point de vue de l'informatique et en dehors de tout contexte applicatif. Or, une application de commande ordonnancée ¨en aveugle¨ selon une de ces méthodes peut exhiber un comportement médiocre, alors qu'un découpage de l'algorithme et une affectation de priorité suivant des notions d'urgence ou d'importance relatives, dépendant du contexte applicatif, peut conduire à obtenir des performances de l'application (stabilité, faible erreur de poursuite...) beaucoup plus satisfaisants. Dans [18] par exemple on montre que l'application basique de Rate Monotonic à un système de commande simple (3 pendules inversés commandés en parallèle) conduit, via la préemption, à de grandes latences de calcul entraînant l'instabilité du système. Un re-découpage du flot de données (cf. section 4) et une affectation des priorités conforme à l'urgence relative des composants permet d'obtenir un fonctionnement satisfaisant (stable pour les 3 commandes). Il est en effet important de minimiser les latences de calcul, ce qui n'est pas (ou mal) pris en compte dans les théories d'ordonnancement citées.

Robustesse et paramètres temporels Les systèmes temps-réel peuvent être classés en systèmes temps-réel ¨durs¨, où l'on interdit le moindre dépassement d'échéance, et les systèmes ¨mous¨ où des dépassements d'échéance occasionnels sont autorisés (sans plus de précision, puisque la littérature sur l'ordonnancement temps-réel ne se préoccupe pas des particularités des applications). Les systèmes de commande sont le plus cités comme exemples de systèmes temps-réel durs, sans approfondir l'argumentation. Or les systèmes de commande en boucle fermée présentent tous une certaine robustesse vis à vis des incertitudes liées

aux paramètres du processus (leur robustesse est due à une marge de phase, qui doit absorber les variations des paramètres autour de leur valeur nominale) [12]. On peut constater expérimentalement (e.g. [8]) qu'ils ont aussi une certaine robustesse vis à vis de variations des paramètres temporels de l'ordonnancement autour de leurs valeurs nominales. On peut même exhiber des systèmes de commande en boucle fermée fonctionnant très bien alors que tous les modules périodiques dépassent leurs échéances en permanence... Les effets constatés lors d'une étude de cas expérimentale rapportée dans [7], où le processus commandé est un pendule inversé et la qualité de la commande est mesurée par la variance de l'erreur d'asservissement sont les suivants (Figure 2) :

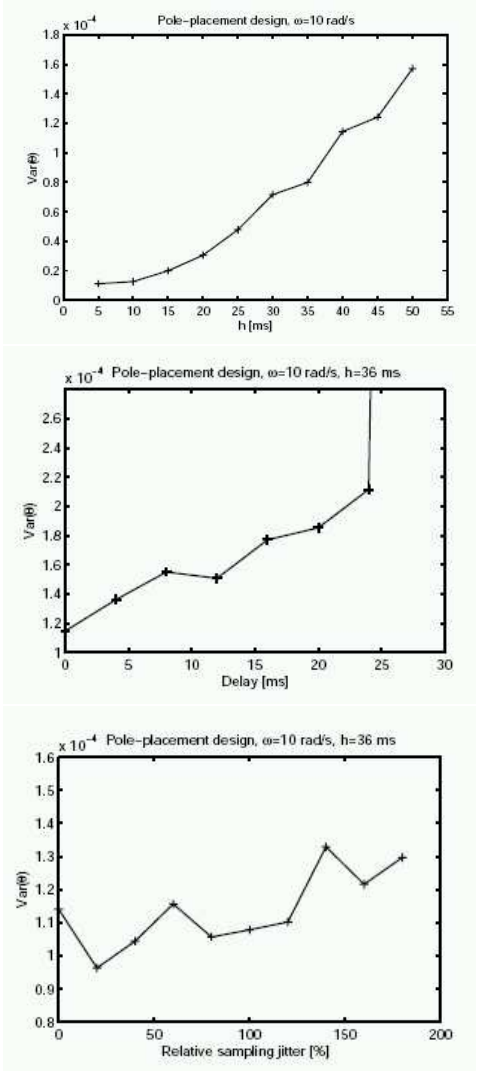


FIG. **2. Effet des variations de période, latence et gigue sur la commande d'un pendule inversé**

- La qualité de commande croît quand la période d'échantillonnage diminue. Cette qualité de commande augmente même si l'on ne profite pas de l'accroissement de la fréquence d'échantillonnage pour augmenter les gains du régulateur [22], [40]. La valeur maximum possible pour la période peut être évaluée grâce à la théorie des systèmes linéaires échantillonnés (théorème de Shannon) ;
- la qualité décroît quand la latence augmente. La détérioration due à la latence croît plus rapidement que celle due à l'augmentation de la période, avec une limite conduisant rapidement à l'instabilité du système (dans l'exemple présenté figure 2 l'algorithme de commande ne tient pas compte du retard ; cette détérioration pourrait être amoindrie avec une commande tenant compte du retard, mais plus compliquée et donc augmentant la période. . .) ;
- la gigue sur l'émission des sorties détériore la qualité, mais l'asservissement reste stable même pour des valeurs de gigue importantes. L'effet de la gigue sur les mesures dépend de l'utilisation qui en est faite. Si l'on mesure directement l´état du système pour l'utiliser dans le calcul de la commande, cet effet n'est pas plus important que celui dû à la gigue de sortie. Cet effet est par contre très sensible si l'on utilise ces mesures dans un processus d'identification ou de reconstruction d'état (imaginer la sortie d'un simple pseudo-dérivateur reconstituant la vitesse à partir des positions par $\hat{\dot{q}_n} = \frac{(q_n - q_{n-1})}{dt}$ si dt n'est pas constant. . .).

4 Conception conjointe commande et ordonnancement

Une première approche consiste à calculer, hors-ligne, une affectation des paramètres d'ordonnancement maximisant (idéalement) la performance de commande du système sous contrainte d'ordonnançabilité des tâches. Le point de départ est bien entendu d'obtenir un modèle de performance fonction des paramètres d'implémentation.

Par exemple, la méthode décrite dans [36] et [37] utilise une fonction de coût entre un critère de performance quadratique et la période d'échantillonnage du contrôleur. Si cette fonction est convexe l'algorithme proposé choisi, parmi toutes les configurations de tâches ordonnançables par la politique choisie (RMA ou EDF) celle maximisant la performance de commande.

Une autre possibilité consiste à formuler le problème d'implémentation de commande sous forme d'un critère de qualité de service rendant compte, par exemple, de la relation performance/période de chaque contrôleur dans le système. Ce modèle peut être utilisé pour configurer le contrôleur d'admission gérant la charge du système [1] ou encore pour négocier en ligne périodes et priorités comme proposé

par [33].

Lorsque plusieurs tâches de commande s'exécutent sur une ressource partagée, la préemption résultant de la concurrence induit des latences de calcul dépendant non seulement de la durée de calcul des fonctions mais aussi de l'entrelacement de leurs instants d'exécution. La méthode décrite dans [31] et [32] utilise des modèles de performance de commande basé sur la théorie des systèmes linéaires, où la fonction de coût décrit la performance (e.g. temps de réponse) du système en fonction des deux paramètres période et retard. Une heuristique itérative d'optimisation ajuste alors les paramètres d'implantation de façon à maximiser la performance globale tout en respectant la faisabilité de l'implémentation. La méthode a été validée hors-ligne mais paraît trop complexe pour être utilisée en temps-réel.

Les approches précédentes supposent que le découpage des algorithmes de commande en tâches est prédéfini : en fait toutes les composantes d'un logiciel de commande n'ont pas la même importance relative si l'on se réfère à leur impact sur une fonction de coût performance/paramètres temporels. Un découpage de l'algorithme de commande conscient de l'urgence et de l'importance relative des composantes permet en particulier de minimiser la latence de calcul sur certains chemins critiques.

Ainsi, il est possible de partitionner le programme du contrôleur d'un système linéaire de la façon suivante [4] :

```
while(1){
Wait-Clock
A/D-Conversion
Calculate-Output /*calcul commande u_k*/
D/A-Conversion
Update-State /*mise à jour modèle x̂_k*/
}
```

où la latence mesure/commande est minimisée en envoyant la commande dès que calculée, la mise à jour du modèle pouvant se faire plus tard, n'importe quand avant la période suivante. Cette méthode est par exemple illustrée dans [18], où ce découpage appliqué à la commande de pendules inversés concurrents apporte une amélioration spectaculaire des performances de commande, sans coût supplémentaire en charge de calcul. Les systèmes non-linéaires peuvent également, et sans doute encore plus, bénéficier d'un découpage adéquat et d'une implantation multi-cadences des processus de commande [39], par exemple en séparant en tâches temps-réel ordonnancées séparément une boucle rapide de stabilisation et une boucle lente de perception-navigation. Un autre exemple concernant la commande de robot est donné dans la section 6.5.

Enfin, du côté informatique des solutions ont été proposées pour prendre en compte les contraintes de la commande dans l'implémentation des contrôleurs. Citons par exemple le modèle des tâches élastiques [6], où l' ¨élasticité¨ des tâches modélise leur sensibilité aux variations de performance (mesure d'un QdS fonction de la période) et permet à l'ensemble de tâches de ¨comprimer¨ harmonieusement leurs périodes pour s'adapter aux surcharges. Citons également le Control-Server [9], où une fraction de la puissance de calcul disponible est réservée à chaque contrôleur : du point de vue de l'exécution sur la ressource partagée chaque contrôleur est isolé, un contrôleur en surcharge temporaire n'a ainsi pas d'impact sur l'exécution des autres. À l'intérieur de chaque segment réservé les tâches sont organisées de façon à minimiser latence et gigue.

5 Ordonnancement régulé (Feedback-scheduling)

Un certain nombre des aspects temps-réel présents dans les systèmes de contrôle/commande ne sont pas convenablement traités par les méthodes classiques d'ordonnancement à priorités fixes. Essentiellement, ce sont :

- l'analyse d'ordonnancement suppose que l'on connaisse les WCET des différentes tâches, ce qui est généralement difficile à obtenir (et même de plus en plus difficile avec les processeurs modernes à multiples niveaux de cache et de pipe-line). De nombreux algorithmes, par exemple en traitement d'image, ont une durée d'exécution dépendant des entrées (nombre de segments dans la scène), inconnue à priori. Tout ordonnancer au temps d'exécution maximum prévisible conduit à une sous-utilisation chronique et importante des ressources de calcul ;
- la charge de calcul peut varier fortement au cours du temps : là encore le traitement d'images et la commande référencée capteurs en robotique en donnent des exemples (traitement dépendant du nombre de capteurs actifs à un instant donné). On peut aussi trouver des exemples dans la gestion de serveurs multimédias [1] ;
- la relation performances/ordonnancement d'une loi de commande est mal connue, surtout quantitativement, et l'affectation des paramètres temporels reste basée sur des résultats non exhaustifs et non généralisables d'expériences et de simulations [40].

L'ordonnancement à priorités dynamiques peut apporter une certaine souplesse en réaffectant en ligne la priorité des tâches en fonction, par exemple, de l'ordre des échéances comme dans l'algorithme EDF [13]. Le coté ¨optimal¨ de l'ordonnancement produit se paye cependant par l'instabilité du système en cas de surcharge (effet domino) et, semble-t-il, par un overhead d'exécution important. Surtout il ne semble pas exister (à notre connaissance) d'O.S. diffusé utilisant un tel ordonnanceur à priorités dynamiques. Enfin, comme déjà mentionné, les décisions d'ordonnancement prises sur un critère purement informatique (maximiser la charge CPU en respectant le maximum d'échéances)

à la fréquence du scheduler ne tiennent pas compte (ou indirectement) des spécificités de l'application. Le fonctionnement d'un tel ordonnanceur peut être amélioré par l'addition d'un processus de supervision, comme le régisseur d'ordonnancement proposé par [16].

Une solution alternative consiste à adapter en temps-réel l'ordonnancement grâce à un régulateur en boucle fermée, par exemple en ajustant la période d'activation de tâches en fonction d'une mesure de qualité de commande (QoC) reflétant les objectifs de performance du système.

L'idée nouvelle, principalement étudiée au laboratoire d'Automatique de l'université de Lund[1], au département d'informatique de l'université de Virginie[2] et aussi récemment en collaboration entre l'INRIA et le LAG [3] consiste à ajuster en boucle fermée les paramètres d'ordonnancement du système en fonction de mesures faites sur celui-ci à l'aide d'un algorithme de commande : on peut alors parler d'*ordonnancement régulé* (ou *feedback scheduling*).

Le principe général de l'architecture est décrit par la figure 3. On trouve bien au niveau de l'ordonnanceur la chaîne mesure/régulateur/actionneur d'une boucle de commande. On y trouve plusieurs processus particuliers :

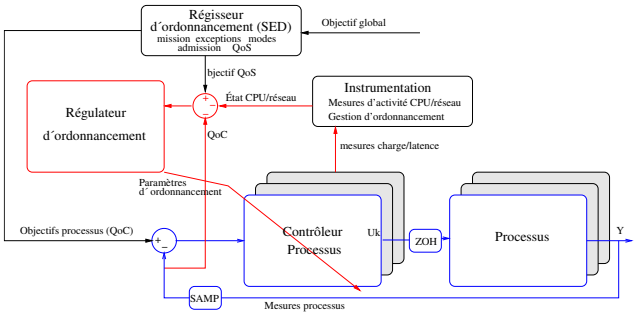


FIG. **3. Ordonnanceur régulé**

- Le *régulateur d'ordonnancement* calcule les paramètres d'ordonnancement (e.g. période, niveau de QoS) en fonction d'un signal d'erreur mesuré sur le processus commandé (e.g. erreur de poursuite) et/ou sur le contrôleur (e.g. charge CPU) ;
- Le *régisseur d'ordonnancement* calcule l'objectif de fonctionnement désiré du système informatique (e.g. consigne de charge CPU), gère l'admission de nouvelles requêtes, traite les exceptions et le passage en mode dégradé...
- Le système doit être *instrumenté* pour permettre d'implémenter ces deux nouvelles tâches ; il s'agit de mesurer l'état et l'activité du système informatique et de manipuler les paramètres d'ordonnancement des contrôleurs de processus. Il peut s'agir d'une couche intergicielle, entre l'O.S. temps-réel et l'application de contrôle/commande.

[1]http ://www.control.lth.se/ anton/artes/
[2]http ://www.cs.virginia.edu/ stankovic/rts.html
[3]http ://www.inrialpes.fr/pop-art/people/simon/c3o/c3o.html

5.1 Capteurs et mesures

Le signal d'erreur est élaboré à partir de mesures prises sur le processus commandé (estimation de QoC) et sur le système informatique. L'état du processus contrôlé est obtenu classiquement à partir de capteurs et d'algorithmes de reconstruction/filtrage.

L'état de la ressource d'exécution peut être obtenu au travers de différentes mesures.

5.1.1 Charge CPU

Une contrainte incontournable dans l'exécution des contrôleurs est la saturation du CPU, se traduisant par des surcharges et des dépassements d'échéances. La charge de calcul induite par les différents modules de programme s'exécutant sur la ressource partagée doit être mesurée ou estimée, cette mesure pouvant être ensuite utilisée par le régulateur d'ordonnancement comme contrainte, comme variable de commande ou encore comme perturbation.

Il faut pour commencer obtenir une estimation $\hat{c}$ de la durée de calcul de chaque tâche temps-réel. Ce n'est possible par mesure directe que si l'OS est instrumenté pour cela (c'est par exemple le cas de RTAI). On peut aussi faire une estimation par le biais du temps de réponse (possible par l'API Posix), qui ne tient pas compte des instants de préemption et donc surestime la charge totale : la synthèse d'un estimateur robuste, simple et non biaisé de la charge CPU à partir des temps de réponse reste à faire...

À partir de l'estimation $\overline{c_i}_{kh_s}$ de la durée de calcul moyenne de la tâche i pendant la kième période h_s de mesure, l'estimation filtrée de la charge induite par cette tâche pendant une période h_s est :

$$\hat{U}_{kh_s} = \lambda\, \hat{U}_{(k-1)h_s} + (1-\lambda)\, \frac{\overline{c_i}_{kh_s}}{h_{((k-1)h_s)}}$$

où λ est facteur d'oubli. L'utilisation des fréquences d'échantillonnage $f_i(kh_s) = 1/u_i(kh_s)$ permet d'obtenir un modèle linéaire de la charge induite par les n tâches de commande périodiques :

$$\hat{U}(kh_S) = \frac{(1-\lambda)}{z-\lambda} \sum_{i=1}^{n} \overline{c}_i(kh_S) f_i(kh_S)$$

.

Un paramètre important est la valeur de la période d'échantillonnage h_s des mesures et du régulateur d'ordonnancement. Cette valeur doit être assez grande (plus grande que tous les h_i des tâches concernées) pour que la mesure obtenue ne soit pas trop bruitée [27]. Ce modèle (dit ¨fluide¨) n'a de sens que si l'on regarde le système ¨d'assez

loin¨, et interdit donc de réagir rapidement à des perturbations soudaines. Le régulateur obtenu devra être complété par un mécanisme de traitement des surcharges transitoires occasionnés par les changements brusques du point de fonctionnement.

5.1.2 Autres mesures possibles

D'autres mesures possibles concernant l'état de la ressource d'exécution sont :

- les dépassements d'échéance sont des événements faciles à détecter ; leur nombre par unité de temps (miss ratio dans [28]) ne donne une mesure non nulle qu'en cas de surcharge du CPU. Ils peuvent cependant être utilisés en complément de l'estimation de charge (cf par exemple 6.1) ;
- la laxité des tâches (temps restant entre la fin d'exécution d'une instance et l'instant d'activation suivant) peut aussi être envisagée comme indicateur de charge du système ;
- dans le cas d'un système distribué, on devra aussi mesurer ou estimer la charge du réseau, les retards induits, les pertes de messages...

5.2 Actionneurs et commandes

Les variables d'action disponibles sont :

- Période des tâches : la charge CPU induite par n tâches de durée c_i et de période h_i est $U = \sum_{i=1}^{n} \frac{C_i}{h_i}$: on voit immédiatement que les périodes sont des actionneurs efficaces pour agir sur la charge globale de calcul. Le système doit fournir des outils ou une API permettant de modifier en ligne les périodes affectées aux tâches ;
- Priorités des tâches : l'ordre des priorités n'affecte pas la charge de calcul mais l'entrelacement des calculs, et donc les latences mesure/commande. Les priorités doivent aussi refléter l'urgence et l'importance relative des composants sur la performance du contrôleur ;
- L'utilisation de variantes {coût d'exécution, performance} d'une même fonctionnalité peut aussi être utilisé comme actionneur sur la charge de calcul, à l'échelle de temps des changements de mode de marche. Ces variantes peuvent être associée à des valeurs de contribution au QoS de l'application ;
- Le régisseur du système est un système à événements discrets gérant globalement pour l'application et la ressource d'exécution partagée l'admission, le rejet ou l'ajournement de nouvelles requêtes d'activation.

6 Exemples de régulateurs d'ordonnancement

Nous allons maintenant donner quelques exemples d'ordonnancement régulé : suivant la complexité de mise en oeuvre et les objectifs de commande choisis les algorithmes de commande sont adaptés à partir de ceux trouvés dans la boîte à outils de l'Automaticien, de la simple commande PID mono-variable à la commande optimale. Devant la difficulté à modéliser précisément le comportement du système, où les incertitudes viennent aussi bien des processus commandés que de la ressource d'exécution, la notion de robustesse est essentielle.

6.1 Commande de serveur par P.I.D.

La formule de base d'un régulateur P.I.D. (Proportionnel-Intégral-Dérivé) est en temps continu [4] :

$$U = K_p.e + K_v.\frac{de}{dt} + K_i.\int_0^t e(\tau).d\tau$$

où U est la commande appliquée au processus et e le signal d'erreur entre la consigne y_d et la sortie mesurée y (figure 4).

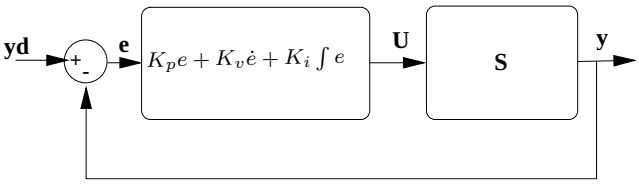


FIG. **4. Boucle de commande**

Il s'agit d'un type de régulateur très répandu car applicable à un grand nombre de processus mono-variables. Il en existe des versions en temps continu (analogiques) et numériques.

Si la réponse en boucle ouverte du processus correspond approximativement à la figure 5, on peut le caractériser par les trois paramètres (signature) L, T et R, ce qui représente un faible effort de modélisation. Un PID est en particulier susceptible de donner des résultats satisfaisants si le rapport T/L est grand (¨retard¨ faible devant la ¨constante de temps¨). Suivant les performances recherchées, la nature du processus et les contraintes de complexité, on peut se contenter d'un régulateur réduit, par exemple P, PI ou PD.

Un technicien un peu expérimenté peut régler un tel régulateur sur le terrain en observant la réponse du système bouclé : le gain proportionnel K_p diminue le temps de réponse, le gain dérivé K_v augmente l'amortissement et gomme les oscillations, enfin le gain intégral K_i annule l'erreur statique.

Il existe d'autre part un certain nombre de méthodes de détermination hors ligne des gains K_p, K_v et K_i : synthèse fréquentielle utilisant la signature, méthode semi-empirique de Ziegler et Nichols, modelage du transfert de boucle [15]... Il est également possible de privilégier la robustesse aux erreurs et variations des paramètres du processus plutôt

que d'augmenter les performances dynamiques en utilisant une méthode de réglage des gains spécifique (PID robuste [14]).

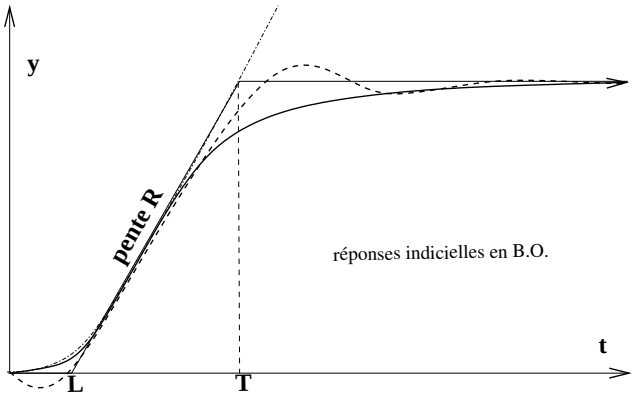


FIG. **5. Caractéristiques en boucle ouverte**

La simplicité conceptuelle du régulateur PID a conduit certains chercheurs a l'utiliser pour commander et réguler le fonctionnement d'un système informatique de type serveur. Cette démarche est en effet bien adaptée à la commande de système à Qualité de Service (QoS), où les contraintes temporelles sont relatives, les dépassements occasionnels autorisés (mais pénalisés) et où la charge (requêtes des clients) est très variable et peu prédictible.

On trouve ainsi le principe de commande en boucle fermée appliqué à l'administration de serveurs web ([2], [26]) ou de serveurs e-mail ([29]). Les principes généraux, des modèles et des études de cas de commande en boucle fermée de systèmes informatiques sont décrits dans [20].

La figure 6 représente le type de d'architecture étudiée dans [27] et [28] détaillée dans cette section.

6.1.1 Modèle du serveur

Le serveur doit exécuter les tâches requises en ligne par les utilisateurs. À chaque tâche est associée une ou plusieurs valeur(s) de QoS, contribuant à la valeur globale du service lorsque la tâche est terminée avec succès, soit avant son échéance. Le but du contrôleur est de maximiser la qualité du service rendu par le serveur, la contrainte étant la saturation du CPU. Dans sa forme la plus générale, le système dispose de plusieurs actionneurs :

- l'ordonnanceur temps-réel est supposé préemptif ; il peut être à priorités fixes et utiliser les politiques Rate Monotonic ou Deadline Monotonic, ou encore à priorités dynamiques et appliquer EDF ;
- le contrôleur de QoS détermine dynamiquement le niveau de qualité auquel va être exécutée chaque tâche admise en fonction de la disponibilité du CPU ;

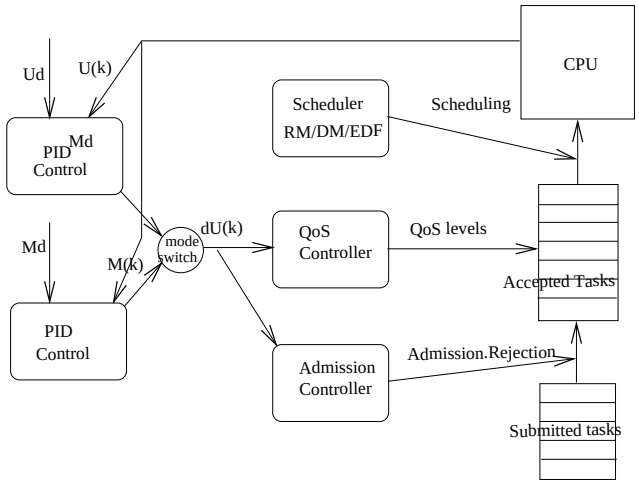


FIG. **6. Contrôle en boucle fermée d'un serveur informatique**

- le contrôleur d'admission admet ou rejette les requêtes incidentes.

Le modèle d'une tâche (pouvant être périodique ou apériodique) T_i est caractérisé par les paramètres suivants :

- N nombre de niveaux de QoS, $N \geqslant 2$;

à chaque niveau j de QoS sont associés :

- $D_i[j]$ échéance relative ;
- $EE_i[j]$ temps d'exécution estimé ;
- $AE_i[j]$ temps d'exécution réel, variable d'une exécution à l'autre et inconnu de l'ordonnanceur ;
- $V_i[j]$ valeur de QoS lorsque la tâche est exécutée avec succès. $V_i[0] \leqslant 0$ est la pénalité payée au QoS global en cas d'insuccès (échéance dépassée ou tâche rejetée par le contrôleur d'admission).

pour les tâches périodiques

- $P_i[j]$ période d'activation
- $B_i[j]$ charge CPU estimée $B_i[j] = EE_i[j]/P_i[j]$
- $A_i[j]$ charge CPU réelle $A_i[j] = AE_i[j]/P_i[j]$

pour les tâches apériodiques

- $EI_i[j]$ intervalle estimé entre deux arrivées de la tâche
- $AI_i[j]$ intervalle moyen (inconnu de l'ordonnanceur)
- $B_i[j]$ charge CPU estimée $B_i[j] = EE_i[j]/EI[j]$
- $A_i[j]$ charge CPU réelle $A_i[j] = AE_i[j]/AI[j]$

6.1.2 Modèle de commande

Les variables mesurées et contrôlées par l'ordonnanceur sont :

- le taux d'échéances dépassées (miss ratio) $M(k)$, défini comme étant le rapport entre le nombre de dépassement d'échéances sur le nombre de tâches exécutées (avec succès ou abandonnées) sur la période $[(k-1)W, kW]$ du contrôleur ;

– l'utilisation du CPU $U(k)$ sur la fenêtre temporelle $[(k-1)W, kW]$, on remarque qu'une saturation du CPU induit automatiquement des dépassements d'échéance ;
– la valeur de qualité totale $V(k)$ sur cette même fenêtre temporelle pourrait être utilisée, en fait cette valeur agit indirectement via l'actionneur de QoS.

La variable d'action est la charge totale estimée $B(k) = \sum_i B_i[l_i(k)]$, sommant la contribution de chaque tâche T_i affectée du niveau de QoS l_i à la kième période d'échantillonage. Elle agit en manipulant les valeurs de QoS l_i affectées aux tâches admises, et également en régulant l'admission de nouvelles requêtes. Sa variation est donnée par $B(k+1) = B(k) + D_B(k)$.

La référence de performance est donnée par des consignes de charge CPU U_S et/ou de taux d'échéances dépassées, par exemple $U_S = 90\%$ *et* $M_S = 0$. L'objectif de commande est donc constitué de deux sous-objectifs complémentaires :

– La commande de charge CPU n'est effective que quand le CPU n'est pas saturé ($U_{real} \leqslant 100\%$). En cas de surcharge une augmentation des requêtes d'exécution ne se traduit plus par une augmentation du nombre de requêtes servies avec succès, mais uniquement du taux d'échecs.
– Quand le CPU n'est pas saturé (et si l'ordonnancement est bien conçu) il ne devrait pas y avoir (ou peu) de dépassements d'échéances. La mesure du taux de dépassement n'est normalement effective qu'en cas de surcharge, lorsque la charge réelle devient supérieure à une charge maximum compatible avec la politique d'ordonnancement (par ex. 1 avec EDF).

Une difficulté prévisible vient de ce que le point de fonctionnement choisi (charge CPU élevée) est proche de la saturation de l'actionneur où le système change radicalement de caractère (en particulier du point de vue de sa commandabilité). On peut cependant montrer que l'utilisation d'un au moins de ces deux actionneurs est toujours possible à un instant donné. Il est donc possible de commander le serveur au moyen de deux contrôleurs mono-variables alternants (fig 6). (On peut aussi utiliser une formulation multi-variables comme dans [17] pour spécifier des objectifs de commande multiples).

On appelle G_A la borne maximum du rapport d'utilisation charge réelle/charge requise ($G_A = max(A(k)/B(k))$) et G_M la borne maximum du rapport entre incrément de dépassements et incrément de charge ($G_M = max(dM(k)/dA(k))$).

Dans les plages non saturées, les modèles linéarisés d'utilisation et de dépassement sont :

$$\begin{cases} U(k) = U(k-1) + G_A.D_B(k-1) \text{ si } A(k) \leqslant 1 \\ \text{ sinon } U(k) = 1 \\ M(k) = M(k-1) + G_M.G_A.D_b(k-1) \text{ si } A(k) > A_{th} \\ \text{ sinon } M(k) = 0 \end{cases}$$

En appelant $X(z)$ la transformée en z de la variable $x(k)$ où z^{-1} est l'opérateur retard ($x_{k-1} = z^{-1}x_k$), on obtient un modèle de comportement du CPU sous forme de fonctions de transfert (en fait de simples filtres passe-bas) :

$$\begin{cases} P_U(z) = U(z)/D_B(z) = \frac{G_A}{(z-1)} \text{ si } A(k) \leqslant 1 \\ P_M(z) = M(z)/D_B(z) = \frac{G_A.G_M}{(z-1)} \text{ si } A(k) > A_{th} \end{cases}$$

On a ainsi obtenu un modèle linéaire échantillonné du comportement du CPU vis à vis de la charge demandée et du respect des échéances. Cette abstraction n'a de sens que si l'on observe de loin le système (qui est par nature un S.E.D.), avec une période d'échantillonnage suffisamment grande pour moyenner et filtrer efficacement l'activité sporadique du système. En pratique la période du régulateur d'ordonnancement doit être très supérieure à toutes les périodes des tâches servies par le système, ce qui interdit *a priori* a l'ordonnanceur régulé de réagir rapidement à une perturbation soudaine. [20] décrit d'autres modèles utilisés pour la commande de systèmes informatiques, dits ¨modèles fluides¨, où par analogie le comportement des files d'attente est assimilé à des réservoirs et celui des contrôleurs à des vannes, les flots d'opérations réalisées par le système étant naturellement modélisés par des débits.

6.1.3 Contrôleur

Le contrôleur doit assurer la stabilité du système, une erreur statique nulle, l'insensibilité aux variations de charge et enfin un temps de réponse et un dépassement de consigne acceptables.

Le régulateur est mono-variable, la commande calculée est une variation de charge D_B ensuite traduite en niveaux de qualité l_i par l'actionneur de QoS (par exemple de type ¨highest-value-density-first¨ [42]).

$$\begin{cases} D_{B_U}(k) = K_{pu}.E_U(k) \text{ où } E_U(k) = U_s - U(k) \text{ si } U \leqslant 1 \\ D_{B_M}(k) = K_{pm}.E_M(k) \text{ où } E_M(k) = M_s - M(k) \text{ si } M > 0 \\ D_B(k) = min(D_{B_U}(k), D_{B_M}(k)) \end{cases}$$

Notons qu'il s'agit d'un simple régulateur proportionnel. Une action intégrale est inutile car présente dans l'actionneur de QoS (ce qui doit assurer la nullité de l'erreur statique). Enfin, compte tenu de l'activité irrégulière du système une action dérivée ne pourrait qu'apporter un niveau de bruit inacceptable.

On peut alors en déduire la fonction de transfert du système bouclé, qui est de la forme :

$$\begin{cases} H_s(z) = \frac{K_p G}{z-(1-K_p G)} \\ Y(z) = H_s(z)\frac{z}{z-1}S \quad \text{(réponse à l'échelon)} \\ G = G_A \text{ ou } G_A G_M \quad \text{(suivant le mode)} \end{cases}$$

Cette structure très simple de régulation ne permet de disposer que d'un paramètre de réglage : le choix d'une valeur particulière de K_p permet de placer (théoriquement) le pôle $p_0 = 1 - K_p G$ du transfert en boucle fermée. L'analyse de ce modèle par les méthodes classiques permet alors de valider simplement les propriétés recherchées de stabilité, d'insensibilité et de robustesse. En particulier, le fait d'avoir calculé le gain K_p en fonction des pires cas largement évalués des facteurs d'utilisation $G_A > G_a(k)$ et de perte $G_M > G_m(K)$ garantit la stabilité du système pour les valeurs réelles des paramètres (robustesse en stabilité). La simplicité du contrôleur se paye par contre par une sensibilité des performances aux variations de paramètres. Enfin, le régulateur d'ordonnancement ne s'occupe pas des priorités, qu'il faut affecter aux tâches selon une politique adéquate.

Les figures 7 et 8 montrent quelques résultats de simulation pour différents profils de charge (détaillés dans [28]) et de politique d'ordonnancement. Le gain du régulateur est choisi pour avoir un temps de réponse à 2% de 4.5 secs (compte tenu du modèle de charge), la période de régulation est choisie empiriquement pour obtenir un compromis acceptable entre stabilité et réactivité.

- DM/PA : tâches périodiques et apériodique, priorités statiques Deadline Monotonic
- EDF/P : tâches périodiques uniquement, priorités dynamiques EDF
- $K_{pu} = 0.185$, $K_{pm} = \begin{cases} 0.414 & \text{si DM/PA} \\ 0.148 & \text{si EDF/P} \end{cases}$
- période du régulateur : 0.5 sec

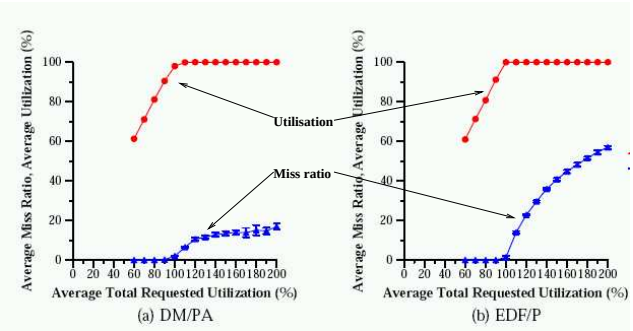


FIG. **7. Réponse en régime permanent**

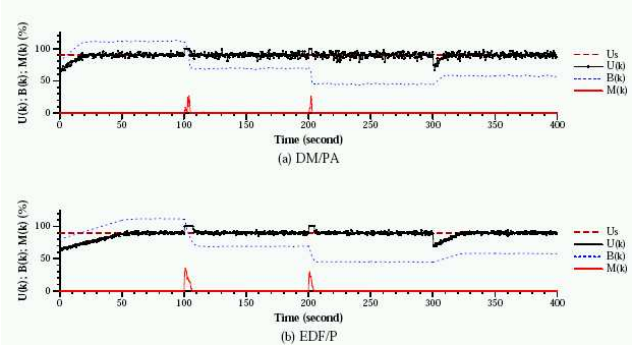


FIG. **8. Réponse transitoire à une surcharge**

On peut observer, en particulier sur la figure 7 donnant le régime permanent (en utilisation et en pertes) fonction de la charge de consigne, l'effet de la politique d'ordonnancement sous-jacente : l'utilisation de EDF permet de reculer jusqu'à $U_d = 1$ le début des dépassements d'échéance, mais en cas de surcharge permanente le comportement du système se dégrade plus rapidement qu'avec la politique à priorité statique DM. La réaction du système à des surcharges transitoires (8) montre le retour des variables contrôlées vers leur valeur de consigne avec une dynamique proche de celle spécifiée.

6.2 Commande optimale

L'approche par régulation de type P.I.D., décrite au paragraphe précédent, a le mérite de la simplicité dans sa conception, son analyse et dans le réglage des gains de commande. Cette simplicité en donne aussi la limite, le nombre très limité de paramètres réglables interdisant, par exemple, de découpler a volonté les transferts du système de commande. Le problème traité était également simplifié puisque le seul processus commandé en boucle fermée était le serveur, les processus contrôlés (e.g. cadencement d'un flot mpeg) étant eux en boucle ouverte.

Revenons au problème initial de commande de processus : le problème consiste à *optimiser* une performance de commande *sous contrainte* d'utilisation bornée de la ressource d'exécution. Ce problème peut être résolu analytiquement dans le cas suivant [19], [8] :

On doit exécuter sur une ressource de calcul partagée n tâches temps-réel, chacune étant le contrôleur d'un système linéaire stochastique ; ces contrôleurs ont pour période h_i et une durée d'exécution C_i. Un critère de performance, dépendant de la fréquence d'échantillonnage, $J_i(h_i)$ est associé à chaque contrôleur i. Il s'agit de maximiser une fonction de coût globale pour l'ensemble des contrôleurs sous contrainte de respect d'un niveau de charge spécifié U_d du calculateur, soit :

$\min_n J = \sum_{i=1}^{n} J_i(h_i)$ sous la contrainte $\sum_{i=1}^{n} C_i/h_i \leqslant U_d$

Le problème a pu être résolu en utilisant pour chacun des contrôleurs une fonction de coût de la forme $J(h) = \frac{1}{h}\int_0^h \left[x^T(t) \quad u^T(t)\right] Q \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt$

On observe sur la figure 9 de telles fonctions de coût en fonction de la période h, correspondant à la commande d'un pendule (inversé à gauche et stable à droite). On constate que cette fonction peut ne pas être convexe, les pics correspondants dans ce cas aux résonances en boucle ouverte du processus commandé (mais ceci est dû au choix particulier de J).

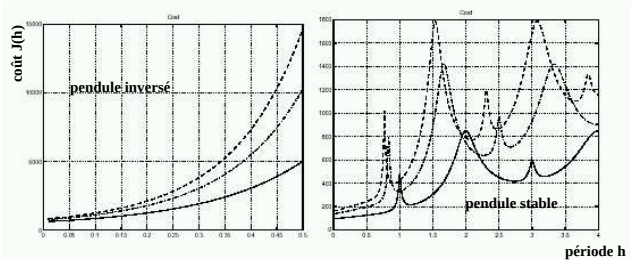


FIG. **9. Exemples de fonctions de coût**

Le choix de cette fonction de coût particulière permet d'obtenir une formulation analytique du contrôleur en boucle fermée réalisant l'optimisation, et pouvant être réalisé sous forme d'un retour d'état. Cette implémentation, nécessitant la résolution en temps-réel d'équations de Lyapunov et de Ricatti, est cependant beaucoup trop coûteuse, en comparaison du coût des contrôleurs de processus eux mêmes, pour être réalisée.

Il existe heureusement des solutions approximatives beaucoup plus simples dans le cas où les fonctions de coût peuvent être approchées par des fonctions quadratiques $J_i(h) = \alpha_i + \beta_i h^2$ ou linéaires $J_i(h) = \alpha_i + \gamma_i h$. Le calcul donnant la valeur optimale des h_i devient particulièrement simple si les fonctions de coût des différents contrôleurs sont *toutes* linéaires ou *toutes* quadratiques [10].

Dans ce cas, le calcul des périodes est obtenu comme suit :

- les valeurs initiales des fréquences de commande $f_i = 1/h_i$ sont choisis en proportion de $(\beta_i/C_i)^{1/3}$ (coûts quadratiques) ou de $(\gamma_i/C_i)^{1/2}$ (coûts linéaires) ;
- ces valeurs correspondent à une charge nominale : $\hat{U_0} = \sum_{i=1}^{n} \frac{\hat{C_i}}{h_{0i}}$
- estimation et filtrage des durées des contrôleurs (filtre passe-bas, λ est un facteur d'oubli : $\hat{C_i}(k) = \lambda \hat{C_i}(k-1) + (1-\lambda)c_i$
- les période des tâches pour d'autres consignes de charge sont données par une simple mise à l'échelle : $h_i = h_{0i}\frac{U_{sp}}{\hat{U_0}}$
- suivant la complexité des contrôleurs, les gains de commande dépendants des périodes h_i sont soit tabulés, soit recalculés en ligne ;
- les consignes de charge totale U_{sp} sont élaborées par un composant de supervision appelé ¨feedforward¨, et jouant en particulier le rôle d'un contrôleur d'admission.

La figure 10 présente quelques résultats de simulation utilisant TrueTime, une boite à outils pour Matlab/Simulink permettant de simuler un modèle de contrôleur incluant les effets de l'implémentation (O.S. temps réel et bus de terrain [11]).

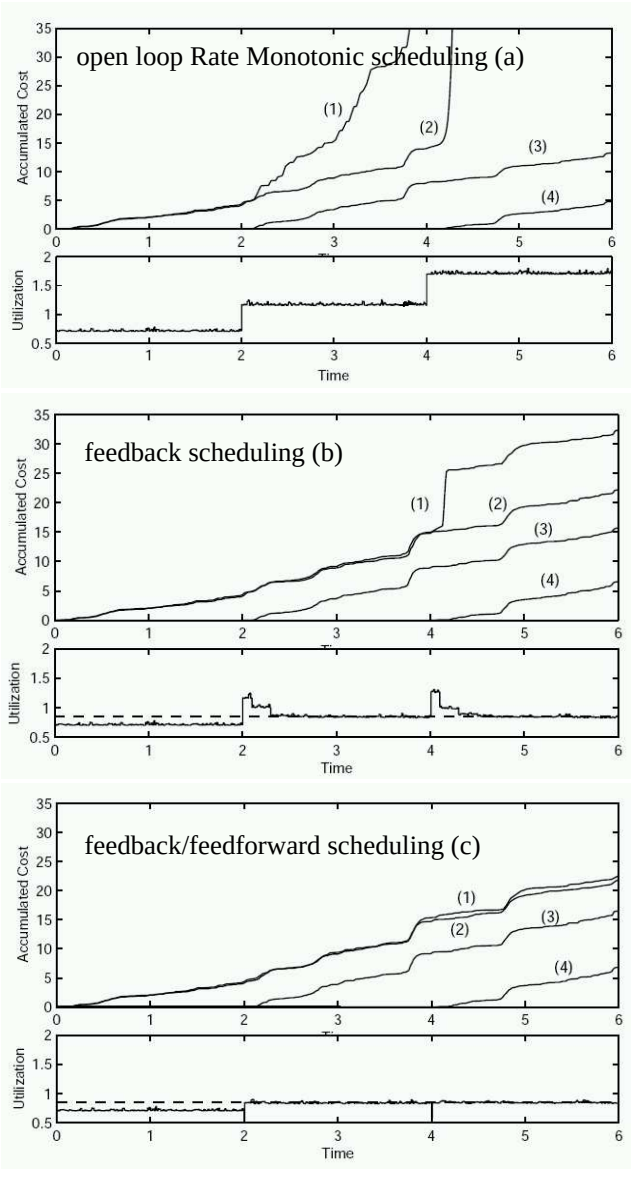


FIG. **10. Simulation TrueTime (commande de pendules)**

Dans cette simulation, on exécute 4 tâches de stabilisation de pendules inversés $T_i, i = 1...4$, dont l'ordre croissant des priorités fixes est $T_1 \prec T_2 \prec T_3 \prec T_4$. Le critère de performance choisi est le coût quadratique classique $J_i = \int_0^{Tsim}(y_i^2(t) + u_i^2(t))dt$. T_1 et T_2 s'exécutent au démarrage du système, T_3 est admise à t = 2 secs et T_4 à t = 4 secs.

Sans adaptation (10a) les contrôleurs s'exécutent à leur fréquence nominale, le calculateur devient surchargé et les tâches les moins prioritaires T_1 et T_2 sont si préemptées qu'elles ne peuvent plus stabiliser leur pendule (le critère devient infini).

L'ordonnancement régulé (10b), en recalculant en ligne les périodes, évite la surcharge du processeur et permet de maintenir tous les systèmes stables (au prix d'une baisse de performance fonction des priorités). L'adjonction d'un contrôleur d'admission (feedforward 10c) permet d'anticiper l'admission de nouvelles tâches et d'améliorer le comportement transitoire de l'ordonnanceur.

6.3 Commande robuste au retard

Les latences dans une boucle de commande sont très perturbantes : elles proviennent des durées de calcul des fonctions de commande, mais aussi de la préemption par des tâches de priorité supérieure au contrôleur en cours d'exécution ; dans le cas d'un système distribué, le réseau est également une source de retards importants (pouvant être supérieurs à la période du contrôleur).

Ces retards provenant de l'implémentation sont difficiles à modéliser précisément, ce qui rend coûteuses ou peu efficaces les méthodes de compensation. Il est plus facile d'estimer une borne supérieure du retard subi : la méthode présentée dans [35] exploite un résultat théorique récent [24] pour synthétiser une commande de processus continu robuste à un retard borné et un ordonnancement régulé par commande H_∞ pour la boucle externe. L'exemple choisi concerne l'exécution de deux tâches de commande concurrentes partageant l'utilisation d'un CPU.

Commande de l'ordonnanceur

- On utilise un modèle normalisé de l'activité CPU : $H(z^{-1}) = \frac{(1-\lambda)z^{-1}}{1-\lambda z^{-1}}$ obtenu par normalisation et transformée en z du modèle de charge linéaire établi section 5.1.1. La valeur de λ (comprise entre 0 et 1) permet de régler la rapidité d'estimation et le niveau de bruit. Les fréquences normalisées calculées par le régulateur via le modèle linéaire sont pondérées par les estimations de durée, écrêtées pour éviter sur et sous-échantillonnage, puis converties en périodes qui sont les actionneurs manipulés par l'OSTR (figure 11).

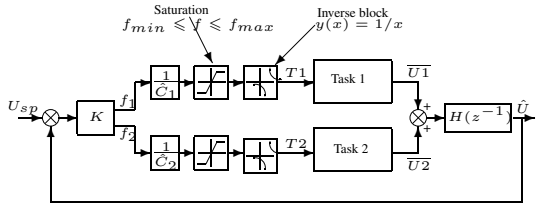


FIG. **11. Ordonnancement régulé de 2 tâches**

- L'utilisation d'un modèle linéaire pour les tâches et l'activité CPU permet d'utiliser la théorie de la commande de systèmes linéaires pour synthétiser le régulateur d'ordonnancement. Le choix d'une méthode de synthèse H_∞ permet de privilégier l'aspect robustesse ([34], [43]).
  Le contrôleur est synthétisé via des gabarits fréquentiels W_i (analogue à celui de la figure 15) permettant de spécifier simultanément une exigence de performance (par exemple le temps de réponse de la boucle de commande de l'ordonnanceur) et une spécification de robustesse (la valeur de la marge de module du système).

Contrôleurs de processus Les processus commandés sont de classiques pendules inversés. Le modèle de commande est $x(k+1) = Ax(k)+Bu(k-d)$ où x est le vecteur d'état, u le vecteur de commande, A et B des matrices de dimensions appropriées et d un délai positif, inconnu mais borné. Ce délai concatène l'ensemble des délais de boucle, provenant des durées de calcul, des préemptions et des communications. Il est montré dans [35] que l'on peut calculer une commande par retour d'état $u(k) = Kx(k)$ permettant de stabiliser le système pour tout délai d tel que $0 \leqslant d \leqslant \bar{d}$. Le problème de modélisation du retard se simplifie donc en estimation d'une valeur maximum. Par contre la performance du système, par exemple en terme de temps de réponse, n'a pas été spécifiée et va en pratique dépendre de la valeur réelle du délai : on a dans ce cas robustesse en stabilité mais pas en performance.

Le contrôleur est synthétisé par une méthode de résolution LMI (Linear Matrix Inequality), les contrôleurs obtenus sont des retours d'état sans mémoire (dont l'ordre est celui du système commandé), peu coûteux à exécuter. Le calcul de leurs gains est par contre très coûteux, il est donc effectué hors ligne pour la gamme prévue de périodes d'échantillonnage et les gains des contrôleurs correspondants sont tabulés en mémoire. On a représenté figure 12 les résultats de simulation comparant une commande classique par placement de pôles et la commande robuste présentée. Les contrôleurs robustes sont tabulés pour la gamme de périodes [0.02-0.5] s par pas de 0.02 s. Le retard maximum $\bar{d}$ admissible par le contrôleur robuste est de 0.05 s, la com-

mande par placement de pôles est synthétisée en négligeant le retard. Le contrôleur H_∞ d'ordonnancement est spécifié pour un temps de réponse de 4 sec. et une marge de module de 0,5 (valeur classique en commande robuste), sa période est fixée à 2 s.

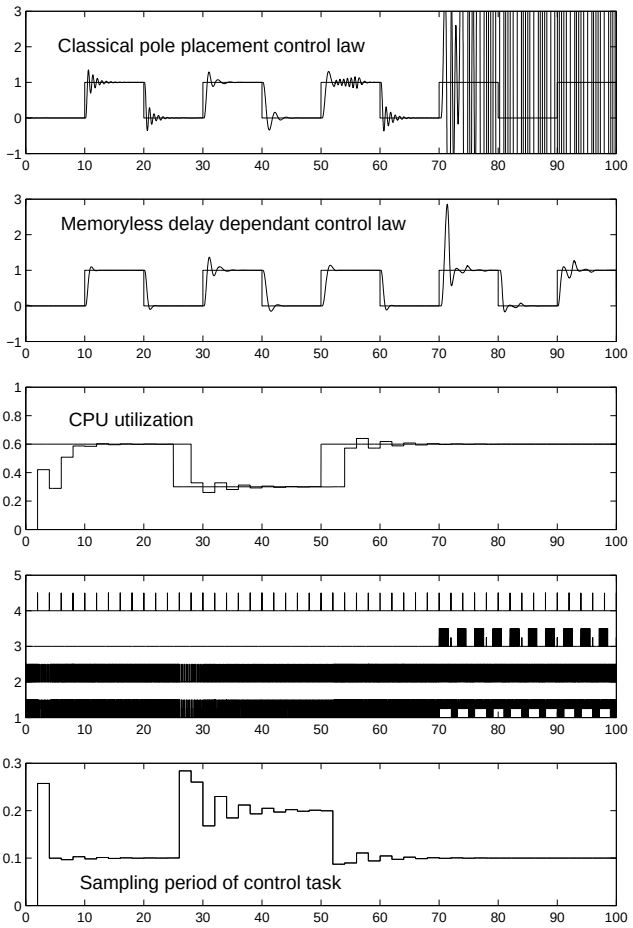


FIG. **12. Feedback scheduling + commande robuste**

En l'absence de retard les deux commandes sont stables. On constate que la charge CPU effectivement utilisée par les contrôleurs suivent la consigne de charge avec le temps de réponse spécifié.

Au temps t=70 s on admet dans le système une tâche perturbatrice prioritaire non mesurée, introduisant par préemption des contrôleurs un retard non compensé. On constate que la commande par placement de pôles est déstabilisée, contrairement à la commande robuste qui subit cependant une dégradation de performance.

On a dans cet exemple utilisé conjointement des résultats de commande robuste aux retards, permettant de contrer l'effet de délais non modélisés induits par l'implémentation, et d'ordonnancement régulé permettant d'adapter aux ressources de calcul disponibles les paramètres d'exécution de lois de commande.

6.4 RST et performances variables

Un défaut de l'exemple précédent est de ne pas permettre de spécifier explicitement les performances des processus contrôlés.

On sait que les performances d'un système de commande varient avec les paramètres d'exécution (périodes, latences et gigue), et que d'une façon générale ces performances sont dégradées quand les valeurs de ces paramètres augmentent. On peut donc supposer que lorsque les ressources d'exécution sont limitées il serait utile de revoir à la baisse les performances désirées pour les rendre compatibles avec un ordonnancement réalisable et conserver la faisabilité de l'exécution en temps-réel. La figure 6.4 compare en simulation le comportement d'un contrôleur sur la gamme de fréquences [2-50] ms : une spécification de performance (temps de réponse) constante aboutit (en haut) à une déstabilisation du système alors qu'une dégradation volontaire de la performance spécifiée, cohérente avec la période réalisable, maintient la stabilité du système.

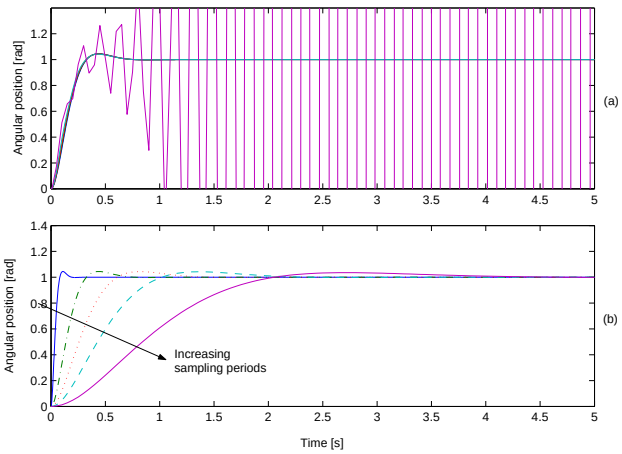


FIG. **13. Performance fixe vs. variable**

Dans [30] on utilise pour cela un régulateur de type RST (une structure générale de commande sous forme polynomiale de système mono-entrée/mono-sortie figure 14). Contrairement au cas classique où le processus est échantillonné à cadence fixe, on va ici considérer que l'on échantillonne avec une fréquence variable . La fonction de transfert en z du régulateur va être explicitement paramétrée par la période d'horloge h. Le modèle échantillonné paramétré par h est de la forme :

$$G(z,h) = \frac{B(z,h)}{A(z,h)} = \frac{b_i(h)z^i + ... + b_0(h)}{z^j + a_{j-1}(h)z^{j-1} + ... + a_0(h)}$$

Pour des systèmes mécaniques tels que les pendules utilisés dans l'exemple proposé l'ordre du système est 2. Les gains du contrôleur sont calculés de façon à ce que le système bouclé se comporte comme un système idéal de même ordre spécifié par :

$$G_{cl}(z,h) = \frac{B(z,h)\,T(z,h)}{A(z,h)\,R(z,h) + B(z,h)\,S(z,h)}$$

La résolution du problème est effectuée sous forme d'équations Diophantine ([4]), résolues de manière analytique en fonction du paramètre h. Pour conserver la stabilité cette paramétrisation implique une variation lente du paramètre h (ce qui est vrai dans ce cas où h varie au plus vite à la fréquence du régulateur d'ordonnancement).

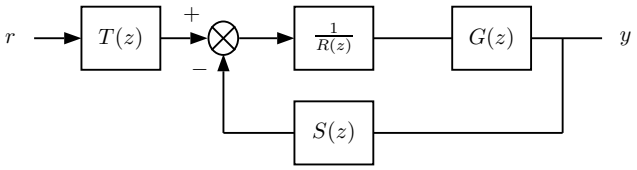


FIG. **14. Régulateur RST**

Spécification de performances Suivant une heuristique classique on choisit pour le contrôleur une période d'échantillonnage h telle que $\omega_{cl}\ h \approx 0.2 \ldots 0.6$, où ω_{cl} est la pulsation la plus rapide présente dans le système bouclé. On introduit ainsi une dépendance entre la performance de commande via ω_{cl} et la période du contrôleur que l'on trouve finalement être de la forme :

$$\begin{aligned} R(z,h) &= (z-1)\,(r_1(h)\,z + r_0(h)) \\ S(z,h) &= s_2(h)\,z^2 + s_1(h)\,z + s_0(h) \\ T(z,h) &= t_2(h)\,z^2 + t_1(h)\,z + t_0(h) \end{aligned}$$

où chaque paramètre est une fraction rationnelle de degré au plus 4, donc encore suffisamment simple pour être calculé en temps-réel.

Régulateur d'ordonnancement Comme dans l'exemple précédent, l'ordonnanceur est régulé par une commande H_∞, utilisant deux gabarits de spécification (figure 15)

- $W_e(s) = \frac{s/M_s+\omega_b}{s+\omega_s\epsilon}$ spécifie le temps de réponse du régulateur d'ordonnancement ;
- W_x spécifie l'allocation du CPU entre les tâches $\frac{U_2}{U_1} \approx \alpha$ modélisant ainsi l'importance relative des contrôleurs.

Les exemples de simulation utilisant cette méthode ([30]) montrent que l'adaptation de la spécification de performance du processus commandé aux variations de la période d'échantillonnage (elle même consécutive à une diminution de la puissance de calcul disponible) permet de gérer une dégradation progressive et contrôlée du système bouclé.

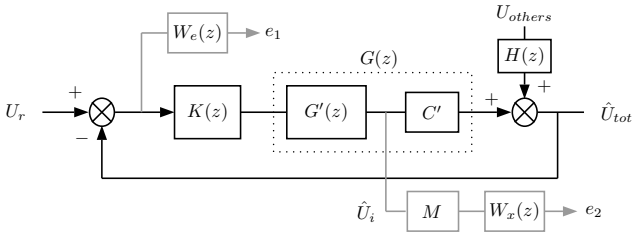


FIG. **15. Régulateur d'ordonnancement H_∞**

6.5 Mise en oeuvre : commande dynamique de robots

Les exemples précédents ont été validés uniquement en simulation, moyennant un certain nombre d'hypothèses, en particulier on suppose connaître un modèle des durées de tâches, par exemple de type constante + bruit de variance connue. On suppose également que le support d'exécution est instrumenté de façon à rendre réalisable la régulation d'ordonnancement.

Nous allons dans ce dernier paragraphe d'évaluer la faisabilité de l'ordonnancement régulé en utilisant des O.S. existants au travers d'un exemple.

Il s'agit d'implémenter la commande en position d'un bras robot (décrite en détail dans [41]).

Le modèle dynamique du bras est

$$\Gamma = M(q)\ddot{q} + Gra(q) + C(q,\dot{q})$$

où Γ est le vecteur des couples de commande, M la matrice d'inertie du bras, Gra le vecteur des forces de gravité et C le vecteur des forces de Coriolis et centrifuges. q et $\dot{q}$ sont les vecteurs des positions et vitesses articulaires du bras, de dimension 7 dans ce cas précis.

La commande non-linéaire (dite Computed Torque Controller) suivante compense par calcul explicite de modèles les variations de la matrice d'inertie ainsi que les forces parasites :

$$\Gamma = \hat{Gra}(q) + \hat{C}(q,\dot{q}) + \hat{M}[K_p(q_d - q) + K_d(\dot{q}_d - \dot{q})]$$

Elle est réalisée sous forme d'un système multi-tâches, avec Γ comme tâche de stabilisation et $\hat{M}$, $\hat{Gra}$ et $\hat{C}$ pour le calcul de termes correctifs (compensation de la dynamique non-linéaire du bras), auxquelles s'ajoute une tâche de génération de trajectoire (figure 17). Toutes ces tâches coopèrent pour la réalisation de la poursuite de trajectoire par le robot tout en étant en concurrence pour l'utilisation du calculateur.

La théorie de la commande n'est pas capable de donner une valeur optimale (si elle existe), ni même raisonnablement bonne, des périodes des tâches minimisant par exemple un critère de performance de suivi de trajectoire.

Notons que, vu le caractère non-linéaire du modèle, le poids respectif de ces périodes devrait varier suivant l'état instantané (position et vitesse) instantané du système commandé. Le but d'un ordonnanceur régulé est dans ce cas d'adapter les périodes des tâches de compensation de façon à obtenir un suivi de trajectoire raisonnablement bon, et d'éviter les overruns malgré une connaissance très approchée des fonctions de sensibilité performance/période et des durées d'exécution des tâches de calcul.

Performance et périodes On utilise comme critère de performance : $J = \int_0^{t_{trajectory}} \sum_{i=1}^{i=7} (q_i - qd_i)^2 dt$, l'intégrale du carré de l'erreur de poursuite sur une trajectoire particulière. Les fonctions de coût sont évaluées en faisant varier séparément la fréquence d'exécution de chacune des 3 tâches de compensation, les 2 autres étant pendant ce temps exécutées à 1KHz. La tâche de stabilisation P.D. (critique pour la stabilité du système) est elle toujours exécutée à une fréquence fixe de 1KHz (figure 16). Notons que, en raison de la nature non-linéaire du système commandé, ces fonctions de coût varient suivant les trajectoires réalisées en terme de positions mais aussi en vitesse de parcours. Pour la trajectoire particulière considérée dans cette expérience on a établi que $J(Iner) \approx J(Coriolis) \approx \frac{1}{2} J(Gra)$.

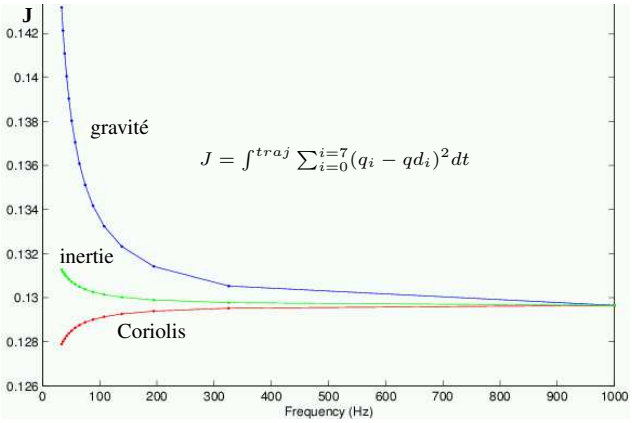


FIG. **16. Fonctions de coût J/fréquence**

Régulateur d'ordonnancement Comme pour l'exemple précédent on utilise une structure de régulateur H_∞ dans lequel les choix de l'utilisateur sont traduits au travers de deux gabarits :

- la fonction de transfert sur le signal d'écart $W_e(s) = \frac{s/M_s + \omega_b}{s + \omega_s \epsilon}$ permet de spécifier la dynamique désirée sur la poursuite de la consigne de charge CPU ;
- les gabarits W_x et M sur la variable ce commande spécifient des poids tels que

$$U_{gravity} = U_{Coriolis} + U_{inertia}$$

, i.e. la fraction de puissance de calcul allouée à chacune des tâches compensatrices est fonction de son importance relative, évaluée à partir des fonctions de coût établies au paragraphe précédent.

Le contrôleur est de la forme $Y_k = CX_k + Du_k$ où u_k est le signal d'erreur (référence - estimation de charge), Y_k (de dimension 3) est le vecteur des fréquences des tâches compensatrices Cor, Gra et $Iner$ et X est le vecteur d'état de l'ordonnanceur (de dimension 4) calculé par $X_{k+1} = AX_k + Bu_k$. A, B, C et D sont des matrices et vecteurs de paramètres constants calculés hors-ligne grâce à la toolbox H_∞ de Matlab.

Le contrôleur résultant est donc de complexité faible. Il est de plus exécuté à une fréquence plus lente que toutes les tâches de commande du robot et sa surcharge induite est donc faible.

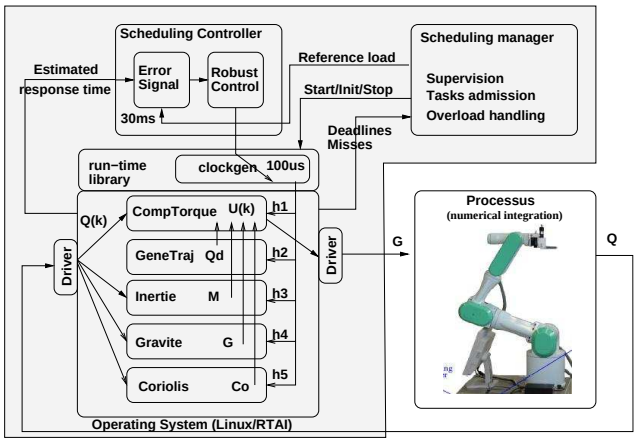


FIG. **17. Architecture de commande de robot**

Faisabilité et architecture Les contrôleurs sont implémentés sur l'architecture multi-tâches décrite par la figure 17. L'ordre décroissant des priorités fixes est déterminé par l'urgence relative des tâches est donné dans la table suivante :

$ClockGen$ (générateur d'horloge 2kHz)
$SchedulingManager$ (event driven)
$SchedulingController$ (période fixe 30 msec)
Γ (période fixe 1msec)
$GeneTraj$ (période fixe 5 msec)
$\hat{Gra} \succ \hat{Cor} \succ \hat{Iner}$ (périodes variables 1-30 msec)

Les priorités ont été affectées en tenant compte de l'urgence relative des tâches, ainsi le contrôleur ayant la période la plus grande est le régulateur d'ordonnancement, mais il contrôle le fonctionnement des contrôleurs de processus, on lui a donc affecté une priorité plus importante qu'à ceux ci. On attribue à la tâche de génération d'horloges la priorité

maximum (dans l'espace utilisateur) de façon à maximiser la régularité des horloges.

L'estimation de la durée d'estimation des modules de calcul est plus ou moins facile et précise suivant l'instrumentation dont est doté l'OS. Par exemple, la mesure est directe avec RTAI (il suffit de lire le champ exectime du descripteur de tâches mis à jour par l'ordonnanceur). Avec Linux et sa librairie de threads Posix (NPTL), il est facile d'estimer le temps de réponse d'un thread mais sans pouvoir y discerner les durées éventuelles de préemption. La norme Posix prévoit une horloge optionnelle CLOCK_THREAD_CPUTIME_ID dont l'utilisation doit permettre une mesure précise et portable de durée d'activité des threads d'une application.

Comparaisons TrueTime et Linux/RTAI Les figures 18 et 19 donnent des résultats de simulation TrueTime (à gauche) et expérimentaux sur un Pentium II 400 MHz (à droite). Le système utilisé est RTAI, permettant d'obtenir une mesure précise des durées d'exécution. À noter que, par mesure de prudence et de disponibilité, le robot (et le robot seul) est simulé : dans ce cas le driver associé à la tâche de commande appelle un intégrateur numérique et un modèle de simulation très complet du robot. Ce driver particulier a donc une durée d'exécution grande et variable, et induit une charge de calcul supplémentaire importante sur le CPU qui doit être dimensionné en conséquence.

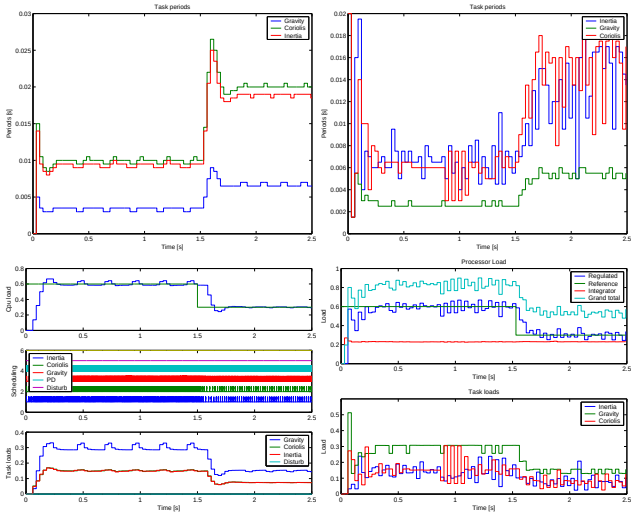


FIG. **18. Périodes et charge CPU**

Les réponses du régulateur d'ordonnancement (périodes et charges mesurées figure 18) diffèrent essentiellement par le niveau de bruit : ce bruit est dû à des variations de durées d'exécution des tâches de calcul (bien que les algorithmes soient tous à nombre d'opérations constants) et à une gigue incompressible des latences d'interruptions (chipset et contrôleur d'ITs...). On peut voir sur la figure 19 que ce bruit n'apparaît pas sur les sorties du processus (positions et vitesses articulaires), celui ci se comportant en filtre passe bas (mais le modèle de simulation ne comporte pas de modes élastiques rapides pouvant être excités par des hautes fréquences).

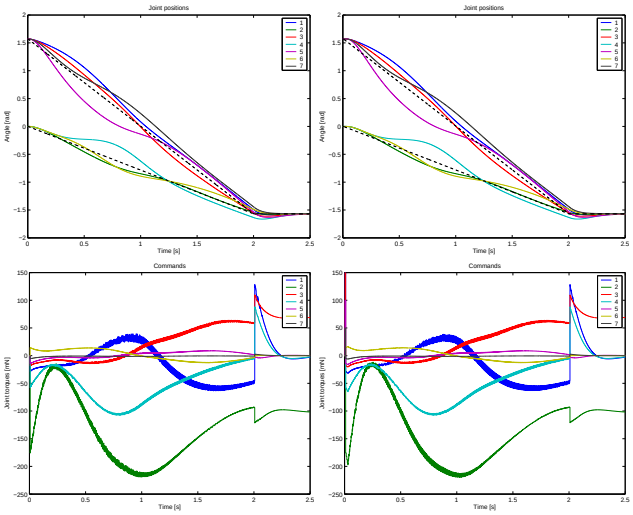


FIG. **19. Position des axes et commandes**

7 Conclusion

Les systèmes de contrôle/commande associent étroitement algorithmes de commande en boucle fermée, logiciel temps-réel embarqué et processus physiques commandés. La conception et l'implémentation de systèmes efficaces et fiables nécessite de prendre en compte des contraintes multiples et parfois contradictoires. La conception conjointe commande/ordonnancement tente de concilier dès le départ de la conception du système les problèmes de performance de commande liés aux contraintes d'implémentation. Si quelques résultats positifs ont pu être obtenus, de nombreux problèmes restent ouverts, en particulier :

- Les relations entre performance de commande et paramètres d'ordonnancement restent mal connus ; si les progrès récents en commande de systèmes à retards donnent maintenant des résultats utilisables en commande robuste, la connaissance du comportement de commandes face à des pertes occasionnelles de données reste très insuffisante pour pouvoir en tirer une formulation en terme de qualité de service et élaborer des commandes robustes et des stratégies de recouvrement d'erreurs adaptées.
- Un système de contrôle/commande distribué peut être adapté vis à vis des perturbations et incertitudes à plusieurs niveaux, pour lesquels des algorithmes efficaces restent à développer : commande spécifique-

ment robuste aux incertitudes temporelles au niveau des contrôleurs de processus (e.g. robustesse aux retards), adaptation dynamiques des paramètres d'ordonnancement au niveau intermédiaire (e.g. feedback scheduling, (m,k)firm policy) et stratégies globales de sûreté de fonctionnement et de gestion de qualité de service au niveau application. Ces différentes stratégies sont l'objet d'un compromis entre diverses contraintes et doivent être développées de façon cohérente.

– Ces stratégies adaptatives agissent en fonction d'observations faites sur l'ensemble du système (processus, calculateurs et réseau) et doivent aussi pouvoir agir sur celui ci. La plate-forme d'exécution, i.e. l'OS temps-réel et le réseau, doit être instrumentés de façon à fournir des mesures et/ou des estimations fiables et précises de l'état du système. Les mesures brutes (e.g. durée d'exécution d'un segment de code) doivent être rendues disponibles au niveau du noyau. Idéalement les mesures plus complexes (e.g. durée d'exécution d'une tâche sur une fenêtre temporelle donnée) et des actions génériques (e.g. modifier la période d'une tâche) devraient être des fonctionnalités portable fournies par une couche intergicielle (ou une API Posix ?).

Références

[1] T. Abdelzaher, E. Atkins, and K. Shin. Qos negotiation in real-time systems and its application to automated flight control. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, june 1997.

[2] T. F. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *39th IEEE Conference on Decision and Control*, Sydney, Australia, december 2000.

[3] K. Arzen, B. Bernhardsson, J. Eker, A. Cervin, P. Persson, K. Nilsson, and L. Sha. Integrated control and scheduling. Technical Report ISRN LUFTD2/TFRT–7686–SE, Dpt. of Automatic Control, Lund Inst. of Technology, august 1999.

[4] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems.* Prentice Hall, 1997.

[5] G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. *IEEE Trans. on Computers*, 50(4) :308–321, 2001.

[6] G. Buttazzo and L. Abeni. Adaptive rate control through elastic scheduling. In *39th Conference on Decision and Control*, Sydney, Australia, 2000.

[7] A. Cervin. Towards the integration of control and real-time scheduling design. Licentiate thesis tfrt-3226, Department of Automatic Control, Lund University, may 2000.

[8] A. Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, Apr. 2003.

[9] A. Cervin and J. Eker. The Control Server : A computational model for real-time control tasks. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 113–120, Porto, Portugal, July 2003.

[10] A. Cervin, J. Eker, B. Bernhardsson, and K. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1–2) :25–53, July 2002.

[11] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. Årzén. How does control timing affect performance ? *IEEE Control Systems Magazine*, 23(3) :16–30, June 2003.

[12] A. Cervin, B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *10th Int. Conf. on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Göteboeg, Sweden, august 2004.

[13] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Ordonnancement temps réel*. HERMES Science Publications, Paris, 2000.

[14] P. de Larminat. La commande robuste : un tour d'horizon. *RAIRO APII*, 25(3) :p 276–296, 1991.

[15] P. de Larminat. *Automatique : Commande des systèmes linéaires*. Hermes Science Publications, 1996, 2ème ed.

[16] J. Delacroix. Stabilité et régisseur d'ordonnancement en temps réel. *Technique et Science Informatiques*, 13(2) :pp223–250, 1994.

[17] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Mimo control of an apache web server : Modeling and controller design. In *American Control Conference*, Anchorage, may 2002.

[18] J. Eker and A. Cervin. A matlab toolbox for real-time and control systems co-design. In *6th International Conference on Real-Time Computing Systems and Applications*, Hong-Kong, december 1999.

[19] J. Eker, P. Hagander, and K.-E. Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12) :pp 1369–1378, 2000.

[20] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[21] D. Henriksson, A. Cervin, J. Åkesson, and K. Årzén. On dynamic realtime scheduling of model predictive controllers. In *41st IEEE Conf. on Decision and Control*, Las Vegas, 2002.

[22] A. Jaritz and M. Spong. An experimental comparison of robust control algorithms on a direct drive manipulator. *IEEE Trans. on Control Systems Technology*, 4(6), November 1996.

[23] C. Kaiser. Système d'acquisition et d'analyse en temps-réel des signaux d'un laminoir rhénalu à neuf-brisach. In *École d'été ETR'99 Applications, Réseaux et Systèmes*, ENSMA, Poitiers, Septembre 1999.

[24] Y. Lee and W. Kwon. Delay-dependent robust stabilization of uncertain discrete discrete-time state-delayed systems. In *FAC 15th World Congress*, Barcelona, Spain, 2002.

[25] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[26] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei,Taiwan, june 2001.

[27] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Real-Time Systems Symposium*, december 2000.

[28] C. Lu, J. Stankovic, S. Son, and G. Tao. Feedback control real-time scheduling : Framework, modeling and algorithms. *Real-Time Systems Journal, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2) :85–126, July 2002.

[29] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Systems Journal*, 23(1-2), 2002.

[30] D. Robert, O. Sename, and D. Simon. Sampling period dependent rst controller used in control/scheduling co-d esign. In *16th IFAC 2005 World Conference*, Prague, july 2005.

[31] M. Ryu, S. Hong, and M. Saksena. Streamlining real-time controller design - from perforamance specifications to end-to-end timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, june 1997.

[32] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In *IEEE Real-Time Systems Symposium*, Madrid, december 1998.

[33] M. Sanfridson. Problem formulations for qos management in automatic control. Technical Report TRITA-MMK 2000 :3, ISSN 1400-1179, ISRN KTH/MMK–00/3–SE, KTH, Stockholm, 2000.

[34] G. Scorletti and V. Fromion. Introduction à la commande multivariable des systèmes : méthodes de synthèse fréquentielle H_∞. `www.greyc.ismra.fr/LAP/Gerard_S/ENSI_comrob.html`, 2004.

[35] O. Sename, D. Simon, and D. Robert. Feedback scheduling for real-time control of systems with communication delays. In *IEEE Conference on Emerging Technologies and Factory Automation*, Lisbon, Portugal, Sept. 2003.

[36] D. Seto, J. Lehoczky, L. Sha, and K. Shin. On task schedulability in real-time control systems. In *17th IEEE Real-Time Systems Symposium*, Washington, december 1996.

[37] D. Seto, J. P. Lehoczky, and L. Sha. Task period selection and schedulability in real-time systems. In *IEEE Real-Time Systems Symposium RTSS '98*, Washington DC, 1998.

[38] L. Sha, T. Abdelzaher, K.-E. årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory : A historical perspective. *Real Time Systems*, 28(2) :101–156, 2004.

[39] D. Simon and F. Benattar. Design of real-time periodic control systems through synchronisation and fixed priorities. *Int. Journal of Systems Science*, 36(2) :57–76, 2005.

[40] D. Simon, E. Castillo, and P. Freedman. Design and analysis of synchronization for real-time closed-loop control in robotics. *IEEE Transactions on Control Systems Technology*, 6(4) :pp 445–461, july 1998.

[41] D. Simon, D. Robert, and O. Sename. Robust control/scheduling co-design : application to robot control. In *RTAS'05 IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, march 2005.

[42] S. Skiena. *The Algorithm Design Manual*. Telos/Springer-Verlag, New York, 1997.

[43] S. Skogestad and I. Postlethwaite. *Multivariable Feedback Control : analysis and design*. John Wiley and Sons, 1996.

[44] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real Time Systems*, 14(3) :219–250, 1998.

Java Temps Réel – un état de l'art

Marc Richard-Foy
AONIX
66/68 Avenue Pierre Brossolette
92247 Malakoff Cedex, France
Marc.Richard-Foy@aonix.fr

Abstract

Ce papier présente un état de l'art de la programmation Java pour les systèmes temps réel. Dans un premier temps on recense les traits du langage Java qui confèrent à celui-ci des caractéristiques intéressantes pour le temps réel. Puis dans un deuxième temps on porte un regard critique sur ceux-ci afin d'en dégager les faiblesses, insuffisances et limitations. Ensuite pour pallier à ces insuffisances on examine les solutions apportées d'une part par la version 5 de Java et d'autre part par la spécification temps réel RTSJ (Real-Time Specification for Java) qui est en phase d'adoption. Finalement on conclut avec l'initiative en cours SCJ (Safety Critical Java) visant à définir un profil Java pour les systèmes critiques qui font l'objet de certification.

1. Introduction

Lorsque le langage Java a été introduit en 1995, il n'était pas envisageable de l'utiliser pour programmer des systèmes temps réel. Le principe fondamental du langage qui vise l'indépendance par rapport aux systèmes d'exploitation et au matériel a délibérément introduit une sémantique faible dans des domaines tel que le comportement des threads, de la synchronisation, des interruptions, de la gestion mémoire et des entrées/sorties.

Cependant la promesse du « Write Once, Run Anywhere » des plateformes Java combinée à l'attrait du langage Java fait que cette technologie prend tout son intérêt dans le monde des systèmes temps réel où les applications sont développées sur de nombreux types différents de processeurs et de systèmes d'exploitation.

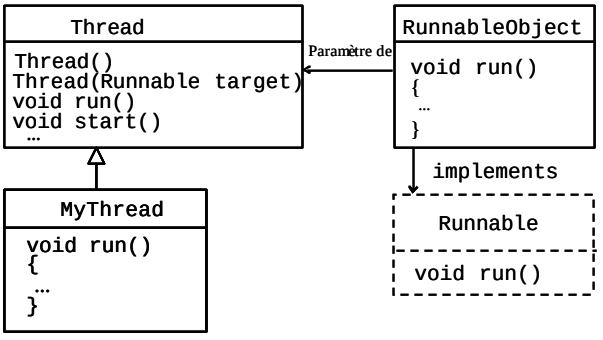
Cet intérêt a motivé la constitution de groupes de travail pour améliorer le modèle de la concurrence Java et pour doter le langage d'une spécification temps réel.

2. Concurrence en Java

Le modèle de concurrence en Java [1] est complètement intégré à l'approche Orienté Objet du langage; elle est basée sur la notion d'objets actifs.

Les threads sont des instances de la classe prédéfinie `java.lang .Thread`. Le corps des threads est encapsulée dans la méthode `run()`. Il y a deux façons de créer les threads. La première consiste à sous classer `java.lang.Thread` tandis que la deuxième consiste à créer un objet `Thread` en lui passant en paramètre un objet `Runnable`.

Lorsque la méthode `start()` d'un thread donné est invoquée, la machine virtuelle [2] déclenche l'exécution de la méthode `run()` de ce thread. Le thread se termine quand sa méthode `run()` se termine. La gestion de la concurrence comprend d'autres méthodes telles que `sleep()` qui permet de suspendre un thread pour un



laps de temps donné ou encore `join()` qui permet de synchroniser des terminaisons de threads.

Le modèle de synchronisation java quant à lui est basé sur le mécanisme du verrou. Chaque objet Java possède un verrou qui est détenu par seul thread à la fois. Ce verrou est utilisé pour l'exclusion mutuelle comme pour la communication asynchrone entre threads. Pour partager un objet en exclusion mutuelle par plusieurs threads il faut que cet objet soit accédé via du code *synchronized* qui se sert du mécanisme du verrou. La communication asynchrone entre threads s'effectue grâce aux méthodes `wait()` et `notify()` qui doivent se synchroniser sur une condition accédée également par du code *synchronized*.

3. Limitations pour les systèmes temps réel

Le modèle multi thread Java bien qu'assez complet présente pour l'essentiel les inconvénients suivants :

- Chaque thread a une priorité qui peut être initialisée dynamiquement dans les limites des valeurs entières de l'intervalle 1 à 10. L'ordonnancement des threads tient compte des priorités des threads, mais cependant il est complètement laissé à l'implémentation. Autrement dit rien ne garantit que le contrôle soit effectivement donné au thread le plus prioritaire lorsque celui-ci devient éligible.

- La méthode `sleep()` permet de programmer des processus périodiques. Cependant l'argument donnant le temps est exprimé en durée ce qui peut donner lieu à des dérives. Il faudrait pouvoir passer en argument un temps absolu (date).

- Le mécanisme des verrous est un mécanisme de bas niveau qui est sujet à l'erreur. Notamment il présente des risques d'inter blocage lorsque des traitements *synchronized* sont interdépendants. Ils nécessitent des styles de programmation implicite pour fonctionner correctement. Par exemple : ne pas accéder directement aux attributs non privés d'une classe qui seraient manipulés par des méthodes qui sont toutes *synchronized*. Un autre exemple est le cas de l'utilisation de la méthode `wait()`. Celle-ci doit toujours être englobée dans une boucle *while* pour garantir que le thread a bien obtenu à nouveau le verrou qu'il a perdu dans la cas d'un `wait()` bloquant.

L'exemple ci-dessous montre un style de programmation à éviter pour l'invocation du `wait()`.

```
Synchronized (p){
   if (!condition){
     p.wait();
   }
   // La condition peut-être fausse !
…
}
```

L'exemple ci-dessous montre à l'inverse le style de programmation qu'il faut adopter.

```
Synchronized (p){
   while (!condition){
     p.wait();
   }
   // La condition est vraie !
…
}
```

- Le déclenchement du Garbage Collector peut bloquer pendant un temps arbitrairement long et imprévisible le système.

- Concernant les traitements asynchrones, Java ne donne pas la possibilité de terminer un thread de façon asynchrone correctement (méthodes `stop()`, `suspend()` et `destroy()` déconseillées). Il ne permet pas non plus la programmation de façon efficace des interruptions (nécessité d'utiliser d'un thread dédié).

4. L'apport de Java 5

Java 5 apporte des améliorations notables du modèle de la concurrence Java. Ces améliorations sont le fruit des travaux menés par les groupes de travail en charge du JSR 133[3] et du JSR 166 [4].

4.1. Le JSR 133

Le JSR 133 améliore le modèle mémoire Java vis-à-vis de la programmation concurrente. Ce modèle définit les conditions nécessaires et suffisantes qui permettent de savoir que les écritures mémoire des autres threads sont visibles par le thread courant et inversement que les écritures par le thread courant sont visibles par les autres threads. La sémantique des constructions qui s'appuient sur les mots clé *synchronized*, *volatile* et *final* a été revue et corrigée pour garantir qu'un programme java correctement synchronisé s'exécute correctement sur toutes les architectures de processeurs quelles que soient les optimisations du compilateur. Notamment, un des soucis majeurs ce cette amélioration a été le contrôle de la réorganisation des séquences d'instructions par les compilateurs ou les processeurs. Un cas bien connu qui illustre cette faille du modèle mémoire Java avant Java 5 est celui de l'idiome controversé « Double-Check Locking » qui vise à éviter le coût de l'utilisation du verrou d'un objet de synchronisation ainsi qu'à différer son initialisation jusqu'au moment de son utilisation. Par exemple, au lieu de programmer la séquence de code ci-dessous dans laquelle le mécanisme de verrou est mis en œuvre à chaque accès de la ressource :

```
public class ServeurRessource {
  private static Ressource ressource = null;
  public static synchronized Resource
     getResource() {
       if(ressource == null) ressource = new
         Ressource();
         return ressource;
     }
}
```

Et aussi afin de différer l'initialisation de la ressource, on utilise la séquence de code ci-après.

```
public class ServeurRessource {
  private static Ressource ressource = null;
  public static Resource getResource() {
    if(ressource == null) {
      synchronized (ServeurRessource.class){
        if(ressource == null)
          ressource = new Ressource();
      }
    }
    return ressource;
  }
}
```

En fait avant Java 5, la construction *synchronized* garantissait seulement l'exclusion mutuelle à l'objet accédé concurremment par plusieurs threads sans se préoccuper de la visibilité des données partagées qui étaient modifiées dans la section de code *synchronized*. Elle ne donnait aucune garantie que l'objet `Ressource` ci-dessus aurait été initialisé avant d'être rendu visible. Dorénavant elle garantit aussi que les données partagées qui sont écrites dans la construction *synchronized* deviennent visibles d'une façon déterministe par les autres threads qui se synchronisent sur ces données dès que le verrou est relâché. Par déterministe il faut entendre « comme si c'était du code séquentiel ».

4.2. Le JSR 166

Le JSR 166 apporte quant à lui un ensemble d'abstractions de haut niveau utiles pour la programmation concurrente. Elles sont contenues dans le paquetage `java.util.concurrent` et elles s'inspirent des composants développés par Doug Lea [5]. Elles constituent une bibliothèque d'utilitaires qui étendent les mécanismes de synchronisation basés sur `wait()` et `notify()` sans les limitations des constructions *synchronized* :

- L'impossibilité de revenir en arrière sur une tentative d'acquisition de verrou qui est déjà détenu par un autre thread, ou encore d'abandonner l'attente après un une période de temps donnée ou à la suite d'une interruption.
- L'impossibilité de personnaliser la sémantique des verrous concernant la réentrance, la protection en lecture/écriture ou l'équité.
- L'absence de contrôle d'accès pour la synchronisation. Toute méthode peut effectuer `synchronized(obj)` pour tout objet accessible.
- L'impossibilité d'acquérir un verrou dans une méthode puis de le relâcher dans une autre.

Les classes qui implémentent les interfaces fournies par le paquetage `java.util.concurrent.locks` constituent le premier niveau d'utilitaires qui lèvent les limitations ci-dessus. Typiquement les objets `lock` s'utilisent comme suit :

```
myLock.lock();
try
{
  Section critique
}
Finally
{
myLock.unlock()
}
```

Cette construction garantit qu'un seul thread peut exécuter la section critique.

Si `myLock` est une instance de la classe `ReentrantLock` alors le thread qui détient le verrou associé peut répétitivement obtenir le verrou qu'il détient déjà. Le verrou maintient un compteur d'appels imbriqués. Grâce à cela le code qui est protégé par le verrou peut très bien appeler d'autres méthodes qui utilisent ce même verrou.

Si au lieu d'invoquer la méthode `lock()` on invoque la méthode `trylock()`, la tentative d'acquisition du verrou est abandonnée si le verrou est déjà détenu par un autre thread. Un délai de garde peut être également précisé ; si ce délai vient à échéance la tentative d'acquisition du verrou est abandonnée.

Si `myLock` est une instance de la classe `ReentrantReadWriteLock` alors il est possible de partager la séquence de code protégée par le verrou par plusieurs lecteurs et un seul écrivain.

L'interface Condition permet d'associer des variables de conditions aux objets `lock`. Ces variables de conditions permettent aux threads qui entrent dans la section de code protégée par le verrou de se synchroniser grâce aux méthodes `await()` et `signal()` d'une façon similaire aux méthodes `Object.wait()` et `Object.notify()` utilisées dans les constructions *synchronized*. Un des avantages par rapport à la celle-ci est de pouvoir associer plusieurs variables de conditions à un seul verrou. Le paradigme du *BoundedBuffer* illustre ce cas de multiples conditions avec l'exemple suivant tiré de la documentation du JDK 1.5 :

```
class BoundedBuffer {
 final Lock lock = new ReentrantLock();
 final Condition notFull = lock.newCondition();
 final Condition notEmpty= lock.newCondition();

 final Object[] items = new Object[100];
 int putptr, takeptr, count;

 public void put(Object x) throws
  InterruptedException {
  lock.lock();
   try {
    while (count == items.length)
     notFull.await();
     items[putptr] = x;
     if (++putptr == items.length) putptr = 0;
     ++count;
     notEmpty.signal();
   } finally {
     lock.unlock();
   }
 }

 public Object take() throws
  InterruptedException {
  lock.lock();
  try {
   while (count == 0)
    notEmpty.await();
   Object x = items[takeptr];
   if (++takeptr == items.length) takeptr = 0;
   --count;
   notFull.signal();
   return x;
  } finally {
    lock.unlock();
  }
 }
}
```

L'interface `BlockingQueue` fournit la notion de file d'attente. Une telle file possède deux opérations fondamentales : ajouter un élément en fin de file et retirer un élément en tête de file. Une file d'attente bloque le thread en cours lorsqu'il tente d'ajouter un élément quand la file est pleine ou de retirer un élément quand la file est vide. Cette interface et les classes qui l'implémentent fournissent des constructions de haut niveau pour programmer des mécanismes producteur / consommateur tout en combinant efficacité et sûreté.

La classe `ConcurrentHashMap` offre des algorithmes sophistiqués pour ne pas verrouiller la totalité d'une table de *hashing* et minimiser les contentions en autorisant les accès simultanés aux différentes parties de la structure de donnée.

Cette classe ainsi que celles qui implémentent l'interface `BlockingQueue` viennent enrichir significativement la structure d'accueil des collections Java existante avec des constructions de haut niveau autorisant la programmation concurrente en toute sécurité.

Un autre apport important de Java 5 est celui de la structure d'accueil `Executor`. Cette structure permet d'invoquer, de planifier, d'exécuter et de contrôler des tâches asynchrones. Ces tâches peuvent être exécutées dans le thread courant, dans un thread en arrière plan, dans un thread nouvellement créé et dédié à cet effet ou encore dans un pool de threads. L'exemple ci-dessous illustre cette structure d'accueil avec l'utilisation d'un pool de thread :

```
Class Service {
  Executor pool =
    Executors.newFixedThreadPool(10);
  public static void main(String[] args){
    Serveur s = new Serveur();
    while (true) {
      final Requete demande = s.accept();
      Runnable r = new Runnable() {
        public void run() {
          traiterRequete(demande);
        }
      } ;
      pool.execute(r) ;
      // au lieu de new Thread(r).start() ;
    }
  }
}
```

On crée tout d'abord un pool de 10 threads qui seront utilisés pour traiter les requêtes. Si la capacité du pool vient à être dépassée la requête est mise en file jusqu'à ce qu'un élément du pool soit disponible pour l'exécuter. L'intérêt du pool est de réutiliser les threads plutôt que de les créer à chaque requête.

Un mécanisme intéressant est celui des tâches de la classe `FutureTask` qui implémente l'interface `Future`. Avec cette interface, il est possible de représenter une tâche qui peut avoir fini son exécution, être en cours d'exécution ou qui n'a pas encore commencé son exécution. A travers cette interface on peut annuler une tâche qui n'a pas encore terminé, demander si la tâche a terminé et récupérer (ou attendre) le résultat de la tâche. Les `Executor` acceptent des tâches qui sont des `Runnable` ou des `Callable` (des `Runnable` dont la méthode run() retourne une valeur). On peut ainsi soumettre en parallèle à un `Executor` des tâches `FutureTask` et récupérer les résultats sans forcément attendre (ce qui représente une certaine forme d'asynchronisme).

Le paquetage `java.util.concurrent` fournit finalement un ensemble de classes qui offre des services de synchronisation tels que les sémaphores, les barrières cycliques, les comptes à rebours et les rendez-vous.

La barrière est un mécanisme qui permet à plusieurs à plusieurs threads de s'attendre à un point commun (la barrière). Quand tous les threads se sont rejoints ils reprennent leur exécution en parallèle jusqu'au prochain passage à la barrière.

Le compte à rebours (`CountDownLatch`) est un mécanisme de synchronisation basé sur une condition qui est fausse au commencement mais qui une fois qu'elle devient vraie le reste pour toujours. Cette condition est représentée par un compte à rebours. Quand ce compte passe à zéro l'ensemble des threads qui attendaient sur cette condition sont débloquées.

Le rendez-vous (`Exchanger`) permet à des threads de s'échanger des objets à des points de rendez-vous. Ce mécanisme est très utile pour les traitements de type producteur/consommateur.

5. La spécification Temps réel RTSJ

Les améliorations introduites dans la version 5 de Java permettent de faire un bond en avant significatif dans la programmation concurrente en offrant des constructions de haut niveau efficaces et sûres, mais cependant elles ne suffisent pas à faire de Java un langage de programmation temps réel. En effet, les nouvelles fonctionnalités du paquetage `java.util.concurrent` ne garantissent rien sur la ponctualité des threads, la prise en compte de leur priorité ou plus généralement leur ordonnancement qui reste tributaire de l'implémentation de l'environnement d'exécution Java sous-jacent. Elles ne donnent aucun moyen particulier pour effectuer l'analyse spatiale et temporelle des systèmes temps réel. Elles ne permettent pas de garantir le déterminisme des systèmes.

C'est l'objectif de la spécification temps réel RTSJ [6] d'apporter une solution à la problématique du déterminisme et de l'analyse temps réel. Cette spécification dont la version initiale a été publiée en 2000 est le résultat du groupe d'experts (RTJEG) en charge du JSR-01 [6]. L'objectif de ce JSR est de répondre aux exigences du NIST [7] qui ont été elles-mêmes l'aboutissement de trois années de réflexions conduites par un groupe de travail indépendant

comprenant des universitaires et des industriels représentatifs de la communauté temps réel.

Le groupe d'experts RTJEG a posé plusieurs principes de base déterminants pour la spécification temps réel :

- Ne pas retreindre la spécification à un type particulier de plateforme comme par exemple Java 2 Micro Edition™.
- Compatibilité en amont avec les programmes Java non temps réel.
- Conformité au principe WORA mais cependant sans altérer le déterminisme.
- Mise en œuvre des techniques temps réel actuelles et à venir.
- Conception privilégiant le déterminisme.
- Pas d'extensions syntaxiques du langage.

Ce dernier principe en particulier a eu un impact important sur la façon dont les mécanismes temps réel ont été fournis. Ceux-ci s'appuient sur des bibliothèques de classes (interface) ainsi que les extensions de la machine virtuelle [TBD] qui leur sont nécessaires. Cette nouvelle interface est contenue dans le paquetage `javax.realtime`. La spécification RTSJ étend Java dans les domaines présentés ci-après.

5.1. Les threads temps réel

Deux nouvelles classes étendent la classe de base `java.lang.Thread` et permettent de définir les threads temps réel :

- La classe `RealTimeThread` définit des threads temps réel qui ont une sémantique d'ordonnancement plus précises que celles de `java.lang.Thread`,
- Et sa sous-classe `NoHeapRealTimeThread` représente les threads temps réel qui n'ont aucune dépendance au tas mémoire. De tels threads ne sont pas autorisés à allouer ni même à référencer des objets du tas mémoire Java. Ces threads peuvent donc s'exécuter en toute sûreté de façon plus prioritaire que le Garbage Collector.

5.2. La gestion mémoire

La spécification RTSJ fournit une gestion mémoire qui ne dépend pas des aléas du Garbage Collector. Elle définit des régions mémoire dont certaines sont en dehors du tas mémoire de sorte que le Garbage Collector n'a pas d'incidence sur celles-ci.

La spécification RTSJ permet que le Garbage Collector puisse être préempté par les threads temps réel pour un temps de latence borné pendant chaque préemption.

Les régions mémoire introduites par RTSJ sont les suivantes :

- `HeapMemory` : c'est le tas mémoire Java classique.
- `ImmortalMemory` : c'est une zone mémoire partagée entre toutes les threads. Les objets alloués dans cette zone ne sont jamais collectés par le Garbage Collector et ne sont libérés que lorsque le programme se termine.
- `ScopedMemory` : c'est une zone pour les objets qui ont une durée de vie bien définie. Un compteur de références associé à chaque `ScopedMemory` permet de conserver la trace de combien d'entités temps réel sont à un moment donné en train d'utiliser cette région mémoire.

Au lancement du programme la région par défaut est le tas classique Java (`HeapMemory`). Ensuite l'invocation de la méthode `MemoryArea.enter()` permet de commuter vers une autre région mémoire. La méthode `MemoryArea.enter()` associe une région mémoire à un objet `Runnable`. Pendant toute la durée d'exécution de la méthode `run()` associée, les allocations sont effectuées dans la région mémoire spécifiée.

Pour les régions de type `ScopedMemory`, lorsque la méthode `run()` se termine, le compteur de références du `ScopedMemory` associé est décrémenté. Si la valeur de ce compteur repasse à 0, les méthodes de finalisation de tous les objets alloués dans la région donnée sont invoquées puis l'espace mémoire associé est récupéré.

L'exemple ci-dessous montre comment s'effectue la commutation d'une région mémoire vers une autre.

```
import javax.realtime.*;
public class Region
{
  // (LTMemory étend la classe ScopedMemory)
  LTMemory locale;
  class Action implements Runnable
  {
    public void run() {
    // Tous les objets alloués dans cette
    // méthode run() ont la portée de
    // l'exécution de cette méthode : ils
    // seront libérés en sortant de la méthode.
     ...
    }
  }
  static public void main(String [] args)
  {
    locale = new LTMemory(8*1024, 8*1024);
    Action action = new Action();
    // Changer de région mémoire
    // Pendant l'exécution de la méthode run()
    // associée à l'argument action les objets
    // seront alloués dans la région locale.
    locale.enter(action);
  }
}
```

5.3. L'ordonnancement des threads.

La spécification RTSJ généralise les entités pouvant être séquencées depuis la notion de threads jusqu'à celle d'objets `Schedulable`. Ces objets implémentent l'interface `Schedulable`, laquelle étend l'interface `Runnable`. La spécification RTSJ fournit deux types d'objets qui implémentent cette interface : les threads temps réel et les handlers d'événement asynchrones.

Lors de leur création, les objets `Schedulable` doivent indiquer quelles sont leurs exigences en terme

d'échéance (moment ou ils deviennent éligibles), de consommation mémoire et d'ordonnancement (priorité). Le gestionnaire de threads programme leur ordonnancement selon ces exigences.

La spécification RTSJ est construite pour permettre aux implémentations de fournir des algorithmes d'ordonnancement non encore connus. Les implémentations doivent être conçues pour permettre d'affecter de façon programmatique les paramètres appropriés pour le mécanisme d'ordonnancement sous-jacent et fournir tous les services nécessaires de création, gestion et terminaison de threads. L'idée est celle d'une structure d'accueil sur laquelle on vient connecter dynamiquement différents algorithmes d'ordonnancement pour des groupes d'objets `Schedulable` donnés.
Par défaut les implémentations doivent fournir l'ordonnancement préemptif par priorité fixe pour au moins 28 niveaux de priorité.

La spécification RTSJ fournit pour caractériser les threads et faciliter l'analyse d'ordonnancement des abstractions intéressantes telles que les classes `SchedulingParameters`, `ReleaseParameters` ou encore `ProcessingGroupParameters`. Ces classes sont utiles pour préciser les attributs des objets `Schedulable` lors de leur construction.

`SchedulingParameters` est une classe abstraite dont dérivent deux sous-classes :

- `PriorityParameters` : Cette classe permet de définir la priorité d'un thread à l'aide d'une valeur entière : plus la valeur est grande plus, la priorité du thread est élevé.
- `ImportanceParameters` : Cette classe permet en cas de surcharge du système de distinguer parmi plusieurs threads de même priorité quel est le plus important.

Les critères de priorité et d'importance des thread sont manipulables via les méthodes `get` et `set` associées.

La classe `ReleaseParameters` définit pour les objets `Schedulable` les attributs suivants :

- La quantité de temps de traitement pour chaque occurrence du thread,
- L'échéance de chaque occurrence du thread,
- L'action à effectuer en cas de débordement du temps de traitement alloué à l'occurrence du thread.
- L'action à effectuer en cas de dépassement de l'échéance

La classe `ReleaseParameters` engendrent les sous-classes `PeriodicParameters`, `SporadicParameters` et `AperiodicParameters`.

La classe `PeriodicParameters` rajoute les deux attributs suivants :

- La date (temps absolu) de la première occurrence du thread,
- La périodicité des occurrences

La classe `SporadicParameters` rajoute l'attribut suivant :

- Durée minimum entre deux occurrences successives,

L'exemple ci-dessous montre comment programmer un thread périodique. Deux manières sont possibles pour

```
import javax.realtime.*;
public class Periodic extends RealtimeThread {

  public Periodic(int priority,
           RelativeTime period,
           Runnable periodicRunnable) {
    super(new PriorityParameters(priority),
     new PeriodicParameters(new
      AbsoluteTime(0,0),
      period,
      null,
      null,
      null,
      null));
    this.periodicRunnable = periodicRunnable;
    }

    public Periodic(int priority,
                         RelativeTime period)
    {
      this(priority, period, null);
    }

    public final void run() {
        while (true) {
            this.handlePeriod();
            this.waitForNextPeriod();
        }
    }

    public void handlePeriod() {
        if (this.periodicRunnable != null) {
            this.periodicRunnable.run();
        }
    };

    private Runnable periodicRunnable;

}
```

créer un Periodic thread : soit en étendant la classe Periodic soit en implémentant l'interface Runnable. Dans les deux cas, le constructeur du `Periodic` thread est invoqué avec les arguments PriorityParameters et PeriodicParameters. Dans ce dernier on ne précise que la date initiale de la première échéance et la périodicité (les autres arguments sont initialisés à `null`).

Le traitement à effectuer à chaque occurrence est encapsulé dans la méthode `handlePeriod()`. Le corps de la méthode `run()` associée au `Periodic` thread comprend une boucle infinie dans laquelle on invoque la méthode `handlePeriod()` fournie par l'application puis la méthode `waitForNextPeriod()` fournie par la spécification RTSJ.

5.4. Le temps et les horloges

La spécification RTSJ définit le temps avec une précision de la nanoseconde et fournit les trois classes `AbsoluteTime`, `RelativeTime` et

`RationalTime` pour manipuler soit des temps absolus (date) ou des intervalles de temps (durées).

Ces différents temps sont relatifs aux horloges. Il est possible d'avoir différents types d'horloge dans un même programme, par exemple une horloge pour mesurer le temps CPU consommé par les threads et une autre pour effectuer des actions temporisées (chiens de garde). Dans tous les cas il y a toujours une horloge temps réel qui est associée à un temps à croissance monotone.

5.5. Synchronisation et partage de ressources

Les mécanismes d'exclusion mutuelles et de partage de ressource proposés par RTSJ sont basés sur les blocs et méthodes synchronisés du Java classique. Cependant pour maîtriser les situations d'inversion de priorité les protocoles d'héritage de priorité transitif et de priorité plafond sont supportés. Les classes `PriorityCeilingEmulation` et `PriorityInheritance` issues de la classe abstraite `MonitorControl` permettent d'assigner une stratégie d'allocation de ressource donnée à un objet donné. Ainsi par exemple, supposons que l'on spécifie le protocole à priorité plafond pour un objet donné. Lorsqu'on y accède via l'instruction `synchronized` et que le verrou associé est passant, la priorité du thread ou de l'objet `Schedulable` qui y accède est montée à la valeur plafond. Dès que le verrou est relâché la priorité repasse à sa valeur précédente.

En plus de ces mécanismes RTSJ prévoit le cas des synchronisations entre les threads non temps réel (`java.lang.Thread`) et les threads temps réel. En particulier ces dernières sont susceptibles de préempter le Garbage Collector. Si la synchronisation est faite selon le protocole d'héritage de priorité il y a le risque de corrompre le tas mémoire géré par le Garbage Collector dans le cas où celui-ci est préempté par le thread temps réel. Pour pallier à ceci, RTSJ introduit des mécanismes de type *Bounded-Buffer* qui permettent à des threads temps réel et non temps réel de se synchroniser et d'échanger des objets dans la région mémoire `ImmortalMemory`.

5.6. Traitement d'événements asynchrones

RTSJ fournit plusieurs moyens qui permettent le contrôle des exécutions asynchrones. Ces moyens sont d'une part le traitement des événements asynchrones et d'autre part le transfert de contrôle asynchrone qui inclue la terminaison des threads temps réel.

Des objets `AsyncEvent` sont utilisés pour faire le lien entre les occurrences d'événements et les traitements asynchrones correspondants. L'occurrence d'un événement peut être initié par le programme d'application, par des mécanismes internes de l'implémentation RTSJ ou encore par sources externes à la machine virtuelle Java comme les interruptions matérielles. Depuis le programme d'application, l'invocation de la méthode `fire()` de la classe `AsyncEvent` permet d'initier l'occurrence d'un événement. Les méthodes `run()` des traitements asynchrones associés à l'occurrence de cet événement sont alors déclenchées. Les traitements asynchrones sont représentés par des objets `Schedulable` qui sont des instances de la classe `AsyncEventHandler`.

Les instances `AsyncEventHandler` se comportent comme si elles étaient exécutées par des threads temps réel. Il n'y a pas forcément un thread temps réel pour chaque `AsyncEventHandler` mais il est cependant possible d'attacher un thread temps réel à un traitement asynchrone donné. La spécification RTSJ impose que la méthode run() d'une instance `AsyncEventHandler` ait conceptuellement la structure suivante :

```
public final void run(){

   while (true) do {
      getAndDecrementPendingFireCount();
      handleAsyncEvent();
      };
    }
```

La méthode handleAsyncEvent() permet d'effectuer le traitement asynchrone correspondant à l'occurrence d'un événement. Cette méthode peut être redéfinit à l'inverse de la méthode `run()` qui est *final*.

`getAndDecrementPendingFireCount()` est une méthode bloquante lorsque le nombre d'occurrences d'événements (la valeur de l'attribut `fireCount`) est nul. Il faut concevoir cette méthode ainsi que son antagoniste `fire()` comme les primitives P et V d'un sémaphore à compte.

5.7. Transfert de contrôle asynchrone

La méthode `interrupt()` de `java.lang.Thread` fournit un moyen de communication asynchrone rudimentaire en positionnant une valeur booléenne dans les attributs de l'objet `Thread`, puis en lançant une exception synchrone lorsque le thread est bloqué sur l'invocation des méthodes `wait()`, `sleep()` ou `join()`. La spécification RTSJ étend l'effet de `Thread.interrupt()` en la redéfinissant pour les threads temps réel dans la classe `RealTimeThread` et en offrant un mécanisme d'exécution asynchrone. Ce mécanisme est basé sur le lancement et la propagation d'exceptions qui quoique asynchrone sont différées quand c'est nécessaire pour éviter de corrompre les structures de données.

5.8. Le statut de RTSJ

La première version de la spécification RTSJ a été publiée en 2000. Une deuxième version faisant suite aux retours d'expérience du groupe d'implémentation de référence a vu le jour en 2002. Depuis lors, cette spécification fait l'objet de commentaires provenant du

Comité d'Interprétation Technique (TIC) qui a publié en juin 2005 la version 1.0.1 qui est en cours d'adoption.

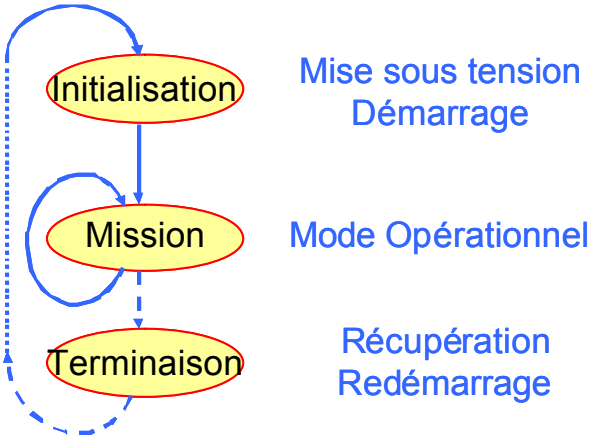
Des idées d'amélioration qui ont été proposées par ce comité [8] feront peut-être l'objet de la soumission d'un nouveau JSR (Java Specification Request).

6. Le profil SCJ pour les systèmes critiques

Un autre domaine concernant Java temps réel dans lequel des groupes de travail sont actifs est celui des systèmes critiques. De tels systèmes doivent se conformer à un processus de certification, comme par exemple dans l'avionique où les logiciels doivent appliquer le standard DO178-B [9]. A l'époque où ce standard a été rendu applicable (1992), les technologies objet n'étaient pour ainsi dire pas utilisées dans les systèmes critiques. Aujourd'hui le déploiement des techniques Orientées Objet est à l'ordre du jour de la révision C du standard DO178.

En anticipation de cette nouvelle révision du standard DO178, outre Atlantique la FAA et la NASA ont constitué le groupe de travail OOTiA (Object Oriented Technology in Aviation) dont l'objectif est de définir des guides [10] pour le déploiement des techniques Orientées Objet pour les systèmes certifiés.

Aujourd'hui parmi les technologies Orientées Objet, Java joue un rôle de première importance, et des projets d'étude tels que Ravenscar-Java[12], Expresso [11] ou HIJA [13] ont entrepris de définir un profil Java pour les systèmes critiques. Ces projets ont en commun l'approche présentée dans le schéma ci-dessous :



Ils supposent que les systèmes critiques s'exécutent en deux temps voire trois temps. Le premier c'est l'initialisation du système, le deuxième temps c'est la mission proprement dite et le troisième correspond à l'arrêt du système. En phase d'initialisation toutes les ressources nécessaires à la mission sont allouées. La mission est constituée d'un ensemble de processus cycliques qui ne s'arrêtent jamais, sauf en cas de défaillance. Dans ce cas la mission est arrêtée et le système redémarre comme indiqué sur le schéma ci-contre.

6.1. Le modèle de la concurrence en SCJ

Le modèle de la concurrence est simple ; il ne comprend que des threads périodiques ou sporadiques. Ceci permet d'effectuer une analyse temps réel en s'appuyant sur les techniques d'ordonnancement préemptif avec priorité fixe. Les threads SCJ sont des threads que l'on peut facilement dériver des threads temps réel fournis par la spécification RTSJ. Plus précisément les caractéristiques du modèle de la concurrence SCJ sont :

- Seuls les objets `Schedulable` globaux (alloués dan la région `ImmortalMemory` en phase d'initialisation) sont supportés,
- Tous les threads temps réel ont des paramètres de déclenchement soit périodiques, soit sporadiques,
- Chaque traitement d'événements asynchrones s'exécute dans un thread temps réel dédié,
- Chaque traitement d'événements asynchrones a des paramètres de déclenchement périodiques, ou sporadiques,
- Chaque traitement d'événements asynchrones est lié à un seul événement bien qu'un événement puisse déclencher plusieurs traitements asynchrones,
- L'ordonnancement est préemptif par priorité fixe,
- L'ordonnancement pour un niveau de priorité donné est FIFO,
- Les dépassements d'échéance sont détectés,
- Le partage de ressource est fait à l'aide de classes avec des méthodes `synchronized`. Aucune suspension n'est permise dans les méthodes `synchronized`,
- L'inversion de priorité est contrôlée par l'utilisation du protocole à priorité plafond.

Toutes les fonctionnalités offertes par la spécification RTSJ et qui sont inutiles pour l'implémentation des exigences ci-dessus sont exclues du profil (exemple le transfert de contrôle asynchrone).

6.2. Le modèle mémoire SCJ

Conformément au schéma ci-contre qui illustre les phases d'exécution des systèmes critiques, Toutes les instances `schedulable` ainsi que tous les objets partagés par ces instances `schedulable` sont alloués dans la région `ImmortalMemory` en phase d'initialisation. Tous les autres objets sont des objets locaux d'instances `schedulable` et ils sont alloués dans la région `ScopedMemory` qui est dédiée à cette instance. Plus précisément les caractéristiques du modèle mémoire SCJ sont :

- La phase d'initialisation s'exécute dan un `ScopedMemory` dédié. Ce `ScopedMemory` a la portée du programme c'est-à-dire jusqu'au prochain redémarrage,

- Les objets alloués en phase d'initialisation ne sont jamais récupérés pendant la durée d'une mission donnée à moins qu'ils ne soient explicitement alloués dans un `ScopedMemory`.
- En phase de mission, seules les allocations dans les régions `ScopedMemory` sont autorisées.
- Chaque instance `Schedulable` a sa propre région `ScopedMemory` où sont effectuées les allocations d'objets lorsque cette instance est déclenchée.
- Aucune allocation dans la région `ImmortalMemory` en phase de mission n'est autorisée.

6.3. Le statut SCJ

Le profil SCJ a été soumis par l'organisme The Open Group en tant que Java Request Specification en 2003. Ce JSR est toujours en phase de définition. Les principaux points techniques en cours de discussion portent essentiellement sur deux sujets. Le premier concerne les moyens d'annotions des programmes qui permettent à des outils d'analyse statique de vérifier les propriétés du profil SCJ. Le deuxième porte sur le modèle mémoire. La question posée étant : faut-il supporter les régions `ScopedMemory` imbriquées ? Actuellement le choix par défaut est de ne pas les supporter.

7. Conclusions

Aujourd'hui Java 5 apporte des moyens de programmation concurrente puissants et robustes avec des abstractions de haut niveau.

La spécification RTSJ qui est en cours d'adoption fournit un cadre à large spectre pour la conception d'applications temps réel calées sur l'état de l'art du moment. Cette spécification peut être elle-même le socle technologique pour la définition de profils couvrant des domaines spécifiques tels que les systèmes critique (SCJ) ou d'autres à venir (systèmes distribués).

L'intérêt d'une spécification java temps réel est de permettre la conception de systèmes temps réel indépendamment de la cible finale ; c'est le concept d'architecture neutre développé par le projet HIJA. L'idée étant de réaliser et valider une fois pour toute le système sur une architecture neutre, puis ensuite de façon incrémentale de valider l'environnement d'exécution Java sur chaque cible où le système est déployé. N'est-ce pas finalement le rêve WORA ?

References

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification", 3ième Edition, http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf, *Addison-Wesley*, 2005.

[2] T. Lindholm, F. Yellin, "The Java™ Virtual Machine", http://java.sun.com/docs/books:vmspec/index.html, seconde edition, *Addison-Wesley*, 2002.

[3] JSR-133, "Java Memory Model and Thread Specification", http://www.jcp.org/aboutJava/communityprocess/review/jsr133/.

[4] JSR-166, "Concurrency Utilities", http://www.jcp.org./en/jsr/detail?id=166.

[5] D. Lea, "Concurrent Programming with Java", Second Edition, *Addison-Wesley*, 2002.

[6] G. Bollella, B. Brosgol, P. Dibble, R. Belliardi, D. Holmes, A. Wellings, "The Real-Time Specification for Java", http://www.rtsj.org/specjavadoc/book_index.html, version 1.0.1, *Addison-Wesley*, 2005.

[7] L. Carnahan, NIST, M. Ruark Commottion Technology, "Requirements For Real-Time Extensions For the Java Platform", http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf, NIST special publication 500-243, 1999.

[8] P. Dibble, A. Wellings, "The Real-Time Specification for Java: Current Status and Future Work", 2004.

[9] "Software Considerations in Airborne Systems and Equipment Certification". DO178-B, RTCA SC-167 / EUROCAE WG-12, 1992.

[10] "Handbook for Object Oriented Technology in Aviation", OOTiA, 2004.

[11] L. Gauthier, M. Richard-Foy, " Realtime Java for Mission and Safety Critical Embedded Systems", http://www.irisa.fr/rntl-expresso/docs/hip-api.pdf, projet RNTL Expresso, 2003.

[12] J Kwon, A. Wellings, R. Steeve, "Ravenscar-Java: a High Integrity Profile for Real-Time Java", University of York Technical Report YCS 342, May 2002.

[13] "HIJA Safety Critical Java Proposal", projet Européen HIJA (High-Integrity Java Application), 2005.

Thème 4

Répartition, Réseaux,
Qualité de Service

Qualité de service dans les réseaux : problématique, solutions et challenges

Zoubir Mammeri
IRIT - Université Paul Sabatier
118 route de Narbonne
31062 Toulouse
mammeri@irit.fr

Résumé

Que les réseaux soient filaires, non filaires, ad hoc, de capteurs, industriels, longue distance, locaux… ils n'échappent pas à une discussion sur leur aptitude à répondre à des besoins de QoS. La prolifération de contributions émanant de laboratoires, industriels, organismes de normalisation, opérateurs de télécoms et fournisseurs d'accès rend difficile la tâche de définir le concept de QoS et de considérer de manière exhaustive des approches sous-jacentes. L'objectif de ce papier de synthèse n'est pas d'être exhaustif dans les fonctions de QoS et encore moins dans les approches et solutions existantes, mais de donner quelques idées sur la manière dont les problèmes sont perçus et traités. Beaucoup de problèmes restent encore ouverts et constituent des challenges dont certains sont soulignés dans ce papier.

1. Introduction

Les débits des réseaux sont passés en l'espace de quelques années de quelques dizaines de Kb/s à plusieurs (dizaines de) Mb/s. Cela permet d'envisager de déployer sérieusement des applications jadis impossible à imaginer à grande échelle. Mais cela pose de nombreux problèmes pour gérer (et garantir) la qualité de service (QoS) requise par ces applications. En effet, de multiples applications (telles que la téléconférence, la commande de processus physiques et la gestion de la localisation de téléphones portables), qui font appel aux réseaux ne peuvent pas fonctionner en faisant abstraction totale de la manière dont le service de communication leur est rendu. Elles sont exigeantes en ce qui concerne le délai de transfert, le taux de perte, la disponibilité des moyens de communications… Ces exigences et d'autres non citées constituent ce que l'on appelle des contraintes de qualité de service (QoS) [2, 9].

Depuis quelques années, beaucoup s'accordent sur les besoins de fournir de la QoS dans les réseaux. Cependant, étant donné la diversité des besoins au niveau applications, des réseaux existants et des coûts que les utilisateurs sont capables de supporter, le nombre de solutions est extrêmement large allant de solutions très statiques (par exemple, réserver une ligne spéciale pour une entreprise), jusqu'aux utilisateurs de l'Internet qui ne souhaitent pas payer (ou payer très peu) pour téléphoner même avec une qualité médiocre.

Pour des raisons économiques, les opérateurs de Télécoms et les fournisseurs d'accès, veulent offrir de la QoS, mais sans bouleversement brutal de leurs infrastructures. Par exemple, avant d'offrir des services de type 3G (dans les réseaux dits de troisième génération), il a fallu passer par des solutions intermédiaires comme le GPRS. Même si techniquement des solutions existent, certaines approches de fourniture de QoS ne sont pas encore disponibles pour des raisons de coûts d'implantation à grande échelle. Un autre exemple est celui de l'Internet où on veut garder le sacro saint IP (pour continuer à fournir un service universel de type 'meilleur effort'), ce qui oblique toute solution d'extension, pour prendre en compte la QoS, de passer par l'utilisation de quelques bits des paquets IP définis il y a longtemps. Cela rend les solutions complexes.

Cependant, malgré la variété des besoins et des solutions, on peut considérer que certains problèmes (du moins dans leur forme générale) se posent presque dans tous les cas : spécification de la QoS et de trafic, gestion de ressources, routage, ordonnancement et contrôle de trafic. C'est à ces aspects que nous consacrerons ce papier. Nous signalons que ce papier n'a pas la prétention ni de poser tous les problèmes liés à la QoS, ni de présenter les différentes solutions apportées à ces problèmes. L'objectif est plutôt d'identifier les problèmes importants et d'esquisser les grandes lignes des approches pour les résoudre tout en signalant les problèmes qui restent ouverts.

Dans la section 2, nous présenterons les concepts de base de QoS et les fonctions de gestion de la QoS. Dans les sections 3 et 4, nous étudierons les problèmes de spécification de trafic et de paramètres de QoS. Ensuite, nous consacrerons une section pour chaque fonction jugée importante, à savoir : contrôle d'admission, ordonnancement de paquets, routage, réservation de ressources et négociation, gestion de buffer et contrôle de congestion, mapping des QoS, contrôle de trafic et de QoS. La section 12 introduit la problématique de la gestion de QoS dans les réseaux mobiles.

2. Définitions et concepts

2.1. Quelques définitions de la QoS

Les préoccupations liées à la QoS sont diverses et la considération de la QoS se fait à plusieurs niveaux : utilisateur, application, middleware, réseau, système d'exploitation, matériel. Par conséquent, il est difficile de trouver une définition qui couvre tous ces niveaux et qui soit unanime. Nous retenons ici deux des définitions les plus citées.

Une définition très générale de la QoS est donnée par l'ITU et ISO [17] : « un ensemble d'exigences de qualité sur le comportement collectif d'un ou plusieurs objets ».

L'IETF, dans son RFC 2216 [21], donne une définition plus axée sur la communication (transfert de données) : « La QoS désigne la manière dont le service de livraison de paquets est fourni et qui est décrite par des paramètres tels que la bande passante, le délai de paquet et le taux de perte de paquets ».

2.2. Niveaux (classes) de service

Les utilisateurs, ou plus exactement leurs paquets[1], subissent, de la part du réseau, un traitement avec un certain niveau de qualité. Pour différencier la qualité du service fourni, on parle de niveau de service. Malheureusement, il n'y a pas encore d'échelle de niveaux de services utilisée et acceptée par tous. Par exemple, dans la 'sphère' des opérateurs et fournisseurs d'accès, on parle de 5 niveaux : premium (délai faible, gigue faible, débit garanti, taux de perte faible), or (délai faible, gigue faible, débit garanti, taux de perte faible), argent (délai non garanti, gigue non garantie, débit garanti, taux de perte garanti), bronze (seul le débit est garanti), meilleur effort (rien n'est garanti).

Dans d'autres communautés (Internet notamment), on distingue trois niveaux de service :

- *Service garanti (ou déterministe) :* la QoS demandée doit être garantie par le fournisseur de service. Ce niveau est généralement exigé par les applications temps réel critiques (strictes).
- *Service probabiliste/statistique* : les paramètres de QoS sont spécifiés par des probabilités ou des contraintes sur la moyenne, variance etc. pour exprimer une certaine tolérance de non respect de la QoS demandée.
- *Service à meilleur effort (*"*best effort*"*) :* le réseau fera de son mieux pour améliorer la QoS fournie mais ne donne aucun engagement pour y parvenir.

Il faut noter que la combinaison de plusieurs critères de QoS rend parfois leur garantie impossible. Par exemple, il est impossible de garantir des délais stricts avec un taux de perte nul (ou plutôt négligeable), car : 1) les réseaux sont sujets à des erreurs inévitables même en surdimensionnant de manière excessive et 2) les retransmissions conduisent au non respect des contraintes de temps.

[1] La notion de paquet est prise ici au sens large. Un paquet peut correspondre aussi bien à un paquet IP qu'à une cellule ATM.

2.3. Classes d'applications et classes de trafic

Différents travaux soutenus essentiellement par des organismes de normalisation ont proposé des classifications des applications ou des trafics en fonction des leurs besoins en termes de QoS, cela permet de faire des correspondances entre les classes d'applications/trafics et les niveaux de QoS offerts par les réseaux. Par exemple, le 3GPP (3rd Generation Partnership Project qui s'occupe des standards liés à l'UMTS) définit quatre classes d'applications :

- Applications *conversationnelles* (par exemple la téléphonie et la téléconférence) : pour lesquelles il faut préserver les relations temporelles entre paquets et garantir un délai faible.
- Applications de type *streaming* : dans ces applications, des flux (audio ou vidéo notamment) sont émis en continu d'un serveur vers un ou plusieurs clients. Il faut préserver les relations temporelles entre paquets et fournir des capacités de synchronisation chez le client par bufferisation (pour éliminer la gigue).
- Applications *interactives* (par exemple les accès au Web et la consultation de bases de données) : pour lesquelles il faut préserver le contenu ; pas de contraintes temporelles strictes mais un temps de réponse optimisé.
- Applications en *arrière-plan* (par exemple, messagerie électronique, transfert de fichier) : pour lesquelles aucune contrainte temporelle n'est spécifiée, mais il faut préserver le contenu.

Dans ATM forum, on parle de classes de trafic : CBR (constant bit rate), rt-VBR (variable bit rate), nrt-VBR (non real-time VBR), ABR (available bit rate), UBR (unspecified bit rate) et GFR (guaranteed frame rate). La garantie des contraintes de temps (délai et gigue) est fournie uniquement par les services CBR et rt-VBR.

2.4. Contrats, SLA

Lorsque le service rendu par le réseau est 'payant' ou contrôlé, un contrat doit être établi et accepté par les deux partenaires : le client (utilisateur) et le fournisseur (le réseau). Ce contrat définit particulièrement les caractéristiques du trafic (débit de crête, débit moyen…) que l'utilisateur génère, les exigences en termes de QoS et éventuellement d'autres aspects (tels que les conditions de pénalité, de ristourne, de résiliation…). Dans la communauté Internet, la notion de SLA (service level agreement) a été introduite pour permettre la spécification de contrat. Sans contrat (explicite ou implicite), il est difficile de parler de QoS.

2.5. Paramètres de QoS

Comme nous l'avions mentionné précédemment, la notion de QoS s'applique à différents niveaux (de l'utilisateur au matériel). Par conséquent, les paramètres permettant de parler et de spécifier la QoS sont divers (qualité sonore d'une bande audio, débit binaire, énergie consommée…). Lorsque l'on s'intéresse uniquement à la communication, les paramètres permettant d'exprimer des besoins de QoS peuvent être regroupés selon plusieurs types [17] :

- *Paramètres temporels* qui incluent notamment : le temps de transfert, le temps de réponse, le temps d'aller-retour, la gigue, le temps d'établissement de connexion et le temps de fermeture de connexion.
- *Paramètres de volume* qui sont spécifiés en terme de débit (binaire, paquets par seconde, transaction par minute…) ou de bande passante.
- *Paramètres d'erreurs* qui incluent notamment : erreur binaire, erreur de paquets, perte de paquets, duplication de paquets, livraison tardive de paquets, livraison désordonnée de paquets, erreur d'établissement de connexion et erreur de fermeture de connexion.
- *Paramètres de fiabilité* qui incluent particulièrement : *MTBF* ("Mean Time Between Failure"), *MTTR* ("Mean Time To Repair") et disponibilité.
- *Paramètres de coûts* qui spécifient le "prix" (en termes d'argent ou autres) que l'utilisateur doit 'payer' (pour chaque flux, pour chaque paquet, par mois…) pour obtenir la QoS demandée.
- *Paramètres d'énergie* : les exigences en consommation d'énergie sont cruciales pour les réseaux sans fil où les opérations d'émission/réception consomment de l'énergie fournie par une source (batterie) limitée et parfois non rechargeable (cas des réseaux de capteurs).
- *Paramètres de sécurité* qui permettent d'exprimer notamment : le degré de protection, le contrôle d'accès, l'authentification et la confidentialité. Il faut signaler que beaucoup ne considèrent pas la sécurité comme de la QoS mais comme un aspect à part entière.

2.6. Natures des métriques

Les paramètres de QoS présentés précédemment sont de natures diverses et leur prise en compte par les fonctions de gestion de la QoS nécessite une maîtrise de la manière de les combiner lorsque l'on veut gérer la qualité de service le long d'un chemin composé de plusieurs composants (éventuellement hétérogènes quant à leur gestion de la OoS). Le premier article à avoir introduit une typologie des métriques est celui de Wang et Crowcroft [22] qui ont distingué trois types de métriques : additive (c'est le cas du délai, puisque l'on additionne les délais des composants intermédiaires pour trouver le délai de bout en bout), multiplicative (c'est le cas de la disponibilité, puisque la disponibilité de bout en bout est le produit des disponibilités des composants intermédiaires) et concave (c'est le cas du débit binaire, puisque le débit de bout en bout est le minimum des débits des composants intermédiaires). Cette classification est un premier pas pour une meilleure prise en compte des paramètres de QoS, mais elle est insuffisante, car comme nous l'avons précisé précédemment (voir aussi §4.1), les contraintes de QoS peuvent être exprimées de manières variées (déterministes ou statistiques). Par exemple, si on un flux qui passe par deux routeurs, un qui garantit un délai compris dans un intervalle connu et l'autre qui garantit que le délai ne dépasse pas une certaine borne avec une certaine probabilité. Dans ce cas, les deux bornes de délais ne sont pas additionnables directement. Dans [18] nous avons proposé un modèle générique permettant de raisonner (à l'aide d'opérateurs polymorphes) sur les paramètres de QoS sous les différentes formes de spécification.

2.7. Gestion de la QoS

Par gestion de QoS, on entend toutes les fonctions qui permettent de prendre en compte la QoS (la fournir, la contrôler, la superviser, la maintenir…). Les fonctions de gestion de la QoS sont nombreuses et prennent des formes plus au moins complexes selon le type de réseau considéré et la nature de QoS à fournir.

Chaque système (contexte) est spécifique et nécessite de rassembler un nombre élevé de 'morceaux' pour former un puzzle. Pour faire l'analogie avec le transport de personnes, pour offrir une qualité de service à un passager voyageant en première classe sur un avion et exigeant une réservation d'hôtel à l'arrivée, on ne fait pas appel aux mêmes fonctions que si cette personne prenait un autocar. Le nombre, la structure et complexité des fonctions de gestion de QoS dépendent des trafics et des niveaux de services qui peuvent leur être fournis.

Pour maîtriser la complexité des fonctions de gestion de QoS, ces dernières peuvent être classées selon plusieurs critères, notamment en distinguant les fonctions liées à des aspects statiques et celles liées à des aspects dynamiques. La première catégorie de fonctions est liée à des propriétés ou exigences qui restent constantes dans le temps. Par exemple, un serveur vidéo demande à transmettre des images à 10 Mb/s. Ces fonctions incluent la spécification des exigences de QoS (i.e. définir ce que l'on attend du réseau), la négociation (i.e. processus qui permet d'arriver à un accord entre l'utilisateur et le réseau), le contrôle d'admission (i.e. le test si la demande de QoS peut être satisfaite), la réservation de ressources (i.e. allocation des ressources pour répondre aux besoins de QoS du flux accepté). C'est la forme la plus simple de gestion de QoS où tout est quasiment prévu d'avance. Dans ce cas, ou bien la

QoS est garantie ou bien les applications se trouvent en situation d'anomalie (voire d'erreur).

La seconde catégorie de fonctions concernent les aspects qui changent dans le temps et qui nécessitent de revoir (renégocier) la QoS fournie par le réseau. C'est la forme la plus complexe de gestion de QoS mais la plus appropriée pour répondre à des besoins variés, changeants et inconnus avec précision à l'avance. Ces fonctions incluent notamment : supervision (i.e. mesurer la QoS réellement fournie et la comparer à celle promise), maintenance (i.e. modifier les paramètres -par exemple le débit alloué à une connexion- du système pour fournir le niveau de QoS requis), renégociation (i.e. revoir certains aspects du contrat pour prendre en compte l'état du réseau et des applications), adaptation (c'est une tâche qui incombe aux applications pour revoir leur fonctionnement afin de tenir compte des capacités du réseau). A noter que la conception d'applications adaptatives est souvent complexe. A titre indicatif, la figure 1 résume l'architecture d'un routeur DiffServ [3] et la diversité des fonctions qui la composent. Dans la suite de ce document, nous reviendrons sur les fonctions de gestion de QoS que nous jugeons importantes.

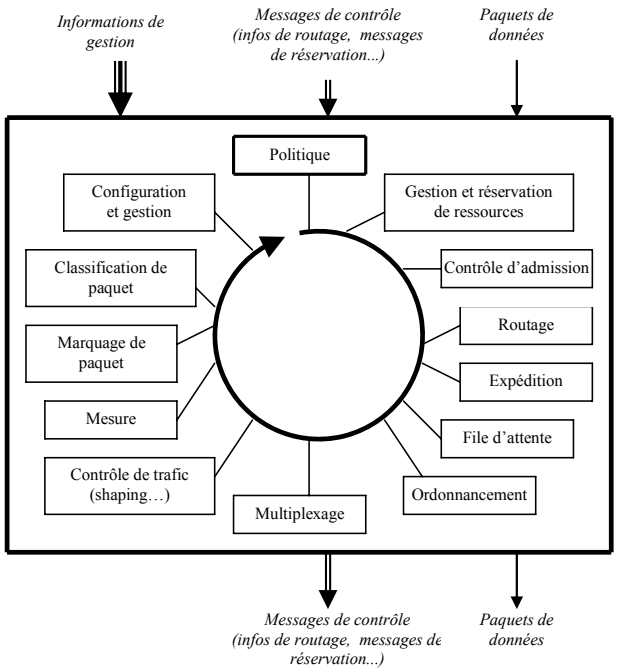


Figure 1. Structure simplifiée d'un routeur DiffServ.

3. Spécification de trafic

Offrir une QoS sans connaissance préalable du trafic est comme conduire un véhicule sans roues directionnelles. Les possibilités de garantie de QoS sont en grande partie liées aux caractéristiques des flux qui traversent les différents équipements composant un réseau. Une spécification précise des trafics permet de mieux réserver et adapter les ressources du réseau pour répondre aux besoins de QoS. Malheureusement, dans beaucoup de cas, les trafics sont spécifiés avec une (grande) imprécision à cause du caractère aléatoire de ces trafics. Il y a deux niveaux se spécification de trafic : spécifier les trafics de flux individuels (par exemple, une source qui transmet un film) et les trafics liés à des flux agrégés (pour des raisons de performance ou de coût, certains équipements ne gèrent pas de flux individuels mais des agrégations de flux).

3.1. Flux individuels (micro flux)

Différents modèles de trafic ont été proposés pour la spécification de trafic ; les plus utilisés dans le cas des flux avec contraintes de temps sont le modèle périodique, le modèle *(Xmin, Xave, I)*, le modèle ATM, le modèle (σ, ρ) et le modèle de l'IETF.

Dans le modèle périodique, le trafic est spécifié à l'aide de deux paramètres : la taille maximale de paquet et la période de transmission.

Dans le modèle *(Xmin, Xave, I)*, trois paramètres sont utilisés pour caractériser un flux avec un comportement avec rafale: *Xmin* (l'intervalle de temps minimum d'inter-arrivée de paquets), *Xave* (l'intervalle de temps moyen d'inter-arrivée de paquets) et *I* (l'intervalle de temps sur lequel *Xave* est calculé, i.e. le long terme).

Dans le modèle de trafic utilisé par le réseau ATM, un trafic avec rafale est spécifié à l'aide de PCR (maximum cell rate), SCR (sustainable cell rate), MCR (minimum cell rate), MBS (maximum burst size), MFS (maximum frame size).

Dans le modèle (σ,ρ), que certains appellent aussi le modèle du seau percé, le trafic est décrit à l'aide de deux paramètres σ et ρ tels que la quantité d'informations transmises par la source durant un intervalle de temps t ne peut pas excéder $\sigma + \rho t$.

Dans le modèle de l'IETF (employé notamment pour fournir de la QoS dans les architectures IntServ et DiffServ), un trafic est spécifié à l'aide de quatre paramètres, taille maximale de paquet (M), débit maximal (P), débit du seau percé (r), taille du seau percé (b) tels que la quantité d'informations transmises pendant t unités de temps ne doit pas dépasser $min(M + Pt, b + rt)$.

Les modèles précédents ainsi que leurs extensions sont largement répandus dans le domaine du trafic temps réel. Ces modèles ont l'avantage d'être simples à décrire donc simples à gérer. En effet, pour accepter un nouveau flux ou pour contrôler si un flux respecte bien son contrat (contrat dans lequel a été faite la spécification du trafic) on se base sur le modèle de trafic. Quand le modèle est simple (avec deux ou trois paramètres), la gestion est simple (car il s'agit le plus souvent de faire des tests à l'aide d'une ou deux équations ou inéquations). Par contre, si le modèle est décrit par de nombreux paramètres et de nombreuses équations, inéquations, probabilités… le modèle devient difficile à gérer. Il y a donc parfois une contradiction à

considérer, en particulier pour les trafics apériodiques (aléatoires de nature) : 1) plus on a d'informations (donc de paramètres) plus on précise la spécification du trafic, 2) moins on a de paramètres, plus on a de facilité à gérer le trafic. La recherche dans le domaine de la spécification de trafic doit prendre en compte cette contradiction. En travaillant plus avec les spécialistes de statistiques, on devrait pouvoir élaborer des modèles proches de la réalité et, peut-être, plus simples à gérer (il y a encore du travail faire).

3.2. Flux agrégés

Les flux individuels peuvent être gérés (i.e. admis et contrôlés) séparément au niveau de tous les équipements le long des chemins qu'ils traversent. Malheureusement, cette gestion présente des problèmes de passage à l'échelle : quand le nombre de flux augmente, le nombre d'informations (états) à gérer et le temps de traitement augmentent en conséquence jusqu'à devenir un obstacle pour le bon fonctionnement du réseau. Pour simplifier la gestion des trafics, certains points du réseau (notamment les routeurs de bordure) procèdent à des agrégations de flux : plusieurs flux sont marqués de la même manière et traités (en terme de QoS) de la même manière ; c'est ce que fait DiffServ par exemple. Si l'agrégation de flux permet la simplification de la gestion de trafic et le passage à l'échelle, elle pose le problème du choix des flux à agréger : quels flux peut-on agréger ensemble ? Ceux qui ont les mêmes caractéristiques et/ou les mêmes besoins de QoS ? Ceux qui se ressemblent avec une certaine marge d'erreur ?... Les réponses à ces questions sont loin d'être évidentes, car d'une part pour simplifier la gestion des trafics, on veut faire appel le plus possible à l'agrégation (à titre d'exemple, dans DiffServ, il y a peu de codes pour marquer les paquets et il faut donc utiliser ces codes avec précaution, donc en faisant appel à l'agrégation le plus possible) et d'autre part, on ne veut pas dégrader la QoS réellement fournie aux flux individuels à cause de l'agrégation.

L'agrégation de flux est encore un sujet peu maîtrisé et du travail reste à faire en particulier pour élaborer :

- des directives et méthodes pour choisir les flux (hétérogènes) à agréger et règles de composition de modèles de trafics individuels ;
- des modèles de perte de précision de caractérisation de trafic en cas d'agrégation (par exemple, si on agrège des flux avec des tailles maximales de paquets différentes, avec des formes de rafales différentes, que représentent les caractéristiques du flux résultant obtenues par rapport à celles des flux individuels ?).
- des modèles de dégradation de QoS fournies aux flux agrégés (par exemple, un flux peut voir son taux de perte augmenter s'il est le plus pénalisé en cas de rejet touchant à l'agrégat auquel il appartient, ce flux 'paye' pour les autres).

4. Spécification de QoS

4.1. Langage de spécification de la QoS

Selon les spécificités des applications (contraintes dures ou souples, coûts à payer, types de réseaux disponibles…), les contraintes de QoS peuvent s'exprimer de plusieurs manières, notamment :

- de manière déterministe en spécifiant soit une valeur qui indique une borne (par exemple, un délai maximum de 10 ms) soit un intervalle (par exemple, un délai compris entre 10 et 15 ms),
- de manière statistique en spécifiant :
  - des conditions sur des paramètres statistiques (par exemple, une variance de délai inférieure à 1 ms),
  - une probabilité (par exemple, un taux d'erreur supérieur ou égal à 99%),
  - une distribution stochastique (par exemple, un délai qui suit une loi gaussienne),
  - un modèle *(m,k)-firm* (par exemple, dans un transfert de vidéo, une image sur 10 peut être perdue),
  - un modèle en logique flou,
  - le meilleur effort (i.e. pas de contraintes pour certains paramètres de QoS).

Une fois les formes de spécification de QoS identifiées, il faut les intégrer à un langage de spécification. Beaucoup de travaux ont été faits pour intégrer la QoS dans des langages comme IDL ou UML ou pour proposer des nouveaux des langages. Dans [1, 18] le lecteur trouvera une synthèse des travaux. Par exemple, on peut définir un contrat de QoS par [13] :

```
Contract_001 = contract {
    numberOfFailures < 10/year;
    TTR {   percentile 100 < 2000 ;
            mean < 500 sec ;
            variance < 0.3 ; };
    Availability > 0.98;
    Delay { mean < 15 ms ; max < 25 ms};
    Throughput { min = 100 Kb/s ;)
```

La QoS est souvent spécifiée de manière ad hoc. La définition d'un langage commun se fait sentir de plus en plus. A notre avis, il s'agit plus de mettre les différents intervenants d'accord sur un langage commun que sur les fondements d'un tel langage.

4.2. Développement orienté QoS

Que ce soit pour développer des applications exigeantes en termes de QoS ou pour développer des services réseau prenant en compte la QoS, il y a un besoin réel d'approches de génie logiciel (depuis la phase d'analyse des besoins jusqu'à la phase d'implantation et test) guidée par les besoins en QoS. Beaucoup de travaux sont proposées pour développer

des applications et services selon l'orientation Composant ou Aspect ou encore Pattern, mais malheureusement sans (ou avec peu de) considération de la QoS. Certains travaux sont actuellement en cours pour intégrer la QoS dans des méthodes basées sur UML ou pour proposer des composants spécifiés par leur QoS. Le génie logiciel guidé par la QoS nécessite encore beaucoup de travaux. Quand on pourra spécifier des composants par les services qu'ils offrent et la QoS qu'ils fournissent, on pourra plus facilement composer des applications et services de haut niveau, faciles à valider du point de vue QoS.

5. Contrôle d'admission

Le contrôle d'admission est le processus qui permet de répondre à la question : est-ce que les contraintes de QoS du nouveau flux peuvent être satisfaites sans remettre en cause celles des flux déjà acceptés ? Ce processus est essentiel pour réguler les flux entrant dans le réseau et garantir la QoS.

Pour accepter ou refuser un nouveau flux, le contrôle d'admission peut utiliser différentes informations :

- la description du trafic (qui peut inclure : taille maximale de paquet, intervalle minimal entre paquets, période, rafales, espacement de rafales, débit moyen, débit de crête, etc.),
- les valeurs des paramètres de QoS demandée,
- l'état et l'historique du réseau (mémoire et bande passante disponibles dans le réseau,...),
- les dates de fin des trafics déjà acceptés,
- les perturbations éventuelles de la QoS des trafics déjà acceptés si le nouveau flux est admis.

Plus les informations (état du système, caractéristiques du trafic à admettre, etc.) sont nombreuses et récentes, plus juste sera la décision d'acceptation ou de rejet des flux et plus coûteux (essentiellement en temps et en échanges de messages) sera le test d'admission.

Il y a deux grandes approches de conception de contrôle d'admission : le contrôle d'admission basé sur les paramètres de flux et le contrôle d'admission basé sur les mesures. Dans la première approche de contrôle, chaque flux doit décrire ses caractéristiques, ensuite le réseau calcule les ressources nécessaires pour l'acceptation de flux. Si les ressources nécessaires au flux sont disponibles, le flux est accepté, sinon il est rejeté. Cette approche est utilisable pour offrir un service garanti (pour les applications temps réel notamment). Dans la seconde approche de contrôle, le réseau se base sur les mesures qu'il effectue (en permanence) sur l'utilisation des ressources pour accepter les flux. Cette approche est utilisable pour offrir un service non garanti mais qui essaie d'améliorer la QoS offerte aux flux.

En général, le contrôle d'admission se fait au niveau du réseau : par chaque nœud du réseau (c'est le cas notamment dans des réseaux IntServ) ou par des nœuds de bordure (c'est le cas notamment dans des réseaux DiffServ). Une troisième approche consiste en : la source envoie des paquets de sonde pour jauger le réseau. En fonction des paquets retour envoyés par le récepteur, la source continue à envoyer son flux normal ou arrêter ; ainsi, la source s'autocontrôle. Cette approche, dite contrôle d'admission par les hôtes ("endpoint admission control"), est préconisée par ceux qui considèrent que le contrôle dans le réseau Internet doit se faire au niveau TCP et non à l'intérieur du réseau (pour éviter de le ralentir) [4]. Cette approche permet d'améliorer le best effort mais ne garantit aucun niveau de QoS.

Les formes de contrôle d'admission sont très nombreuses, elles dépendent des modèles de trafic à gérer et des niveaux de QoS qui peuvent être fournis. Nous allons présenter brièvement deux exemples de contrôle d'admission.

5.1. Contrôle pour QoS déterministe

Pour garantir une borne (de délai, par exemple), on doit spécifier les paramètres de trafic notamment le débit de crête, cela permet à chaque nœud de tester si la somme des débits de crêtes des flux ne dépasse pas la capacité de ses liens de sortie. A titre d'exemple, nous présentons un cas simple de contrôle d'admission pour des flux qui demandent des délais bornés pour un seul nœud, ce nœud n'a qu'un seul lien de sortie. On suppose que chaque flux i est défini par : le débit moyen (ρ_i), l'intervalle de temps T sur lequel le débit moyen est calculé, P_i le débit de crête et D_i le délai demandé par la source du flux i au noeud.

On calcule la quantité maximale de bits, $b_i(\tau)$, que peut transmettre la source du flux i pendant tout intervalle de temps égal à τ par :

$$b_i(\tau) = \min\{\lceil p_i(\tau \mod T)\rceil, \lceil \rho_i T\rceil\} + \left\lceil \frac{\tau}{T} \right\rceil \lceil \rho_i T\rceil$$

Le délai d'attente maximum au niveau du nœud est :

$$D = \frac{\max_{\tau \geq 0} \left\{ \sum_{i=1}^{n} b_i(\tau) - rT \right\}}{r}$$

où n désigne le nombre de flux traversant le nœud et r la capacité du lien de sortie du nœud.

Le test d'admission est : $D \leq D_i$.

On voit que sur cet exemple simple que le coût (en calcul) du test n'est pas négligeable.

5.2. Contrôle pour QoS statistique

Ce type de contrôle est adapté pour la gestion de flux dont la spécification se fait manière statistique. Cela

nécessite donc une connaissance sur les lois d'arrivée des paquets. En utilisant les distributions statistiques des flux, chaque nœud (ou l'ensemble du réseau) garantit une borne, avec une certaine probabilité, pour le paramètre de QoS considérée. Si on se limite au taux de perte de paquet, alors on a la probabilité suivante :

$$\Pr\{(Tagr - Bpd)\tau > buffer\} \leq \varepsilon$$

où *Tagr* désigne le flux agrégé (correspondant au multiplexage des sources), *Bpd* la bande passante disponible, τ un intervalle de temps et ε le taux de perte souhaité. Lorsque un flux arrive, on teste si la probabilité décrite ci-dessus peut être satisfaite ou non.

Chaque flux est modélisé par une loi de distribution (Bernouilli, Gauss, Poisson...). Le test d'admission porte sur l'estimation de *Tagr* et est spécifique à chaque loi de distribution. Par exemple, si *n* flux sont définis par des lois gaussiennes, alors l'approximation du flux agrégé, *Tagr*, peut se faire par une loi gaussienne. Dans [14], les auteurs proposent une équivalence en terme de débit, pour le flux agrégé comme étant égal à :

$$\left(\sum_{i=1}^{n} \rho_i\right) + \left(\left(\sqrt{\sum_{i=1}^{n} \sigma_i^{\,2}}\right)\left(\sqrt{-2\ln(\varepsilon) - \ln(2\pi)}\right)\right)$$

où ρ_i et σ_i sont les paramètres de la distribution gaussienne du flux *i (i=1,..., n)*.

Ensuite, se pose le problème de l'estimation de la bande passante disponible (*Bpd*). Cette estimation est basée soit sur un pire cas d'utilisation des ressources, soit sur des mesures, on parle dans ce cas de contrôle d'admission basé sur les mesures [5]. Cette dernière approche est très efficace pour améliorer la QoS fournie (sans garantie), mais qui est complexe à mettre en œuvre, car elle nécessite une modélisation et analyse statistiques.

Des travaux restent à faire pour avoir des tests de contrôle d'admission à faible complexité et qui tiennent compte des modèles de trafics réels. La maîtrise des phénomènes stochastiques est plus que jamais nécessaire pour élaborer de tels tests.

6. Ordonnancement de paquets

6.1. Concept et propriétés de l'ordonnancement

Les paquets transitant entre une source et une destination passent par un ensemble de nœuds de commutation (tels des routeurs IP, commutateurs ATM, points d'accès Wifi…). Pour simplifier, on peut considérer que chaque nœud de commutation peut être modélisé par un ou plusieurs liens d'entrée et un ou plusieurs liens de sortie (figure 2).

Chaque lien a sa propre vitesse de transmission et a une file d'attente. C'est l'algorithme d'ordonnancement qui gère les files associées aux liens de sortie et qui sert les paquets selon leur priorité. L'ordonnancement de paquets est une clé de voûte pour la garantie de QoS ; il doit permettre aux différentes classes de trafic d'obtenir la QoS qui leur a été promise au moment de leur acceptation.

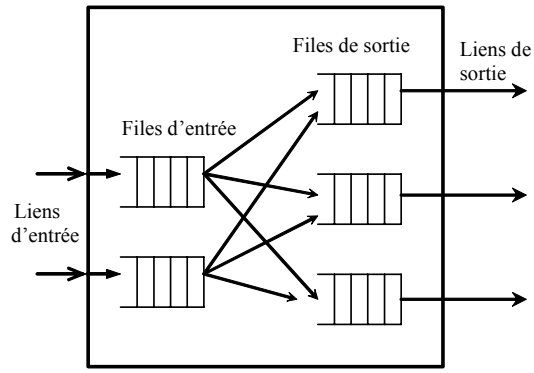


Figure 2. Structure simplifié d'un nœud de communication.

Dans le domaine des réseaux, on parle de manière interchangeable de *politiques d'ordonnancement* ou de *politiques de service*.

Les politiques d'ordonnancement se distinguent tout d'abord sur leurs champs d'utilisation :

- En fonction des contraintes[2] de QoS (délai, gigue, taux de perte, disponibilité…) qu'elles peuvent prendre en compte.
- En fonction de la prise des décisions d'ordonnancement (décisions locales ou globales).

Ensuite, les politiques se distinguent par leurs propriétés. Une politique d'ordonnancement doit avoir certaines propriétés pour envisager son utilisation dans un contexte temps réel, notamment :

- Isolation (ou protection) des flux de manière à ce qu'un flux obtienne une QoS indépendamment du comportement des autre flux.
- Garantie de faibles délais de transfert de manière à pouvoir répondre aux exigences des applications.
- Optimisation de l'utilisation des ressources, notamment la bande passante doit être utilisée au maximum (en évitant l'oisiveté des liens).
- Acceptation d'un nombre élevé de trafics (ou connexions) simultanés.
- Equité : la bande passante disponible doit être répartie de manière équitable entre les flux (malheureusement, l'équité est contradictoire avec la notion d'importance des flux temps réel).

[2] Il est important de souligner que la prise en compte simultanément de plusieurs types de contraintes de QoS rend la politique d'ordonnancement très complexe à analyser.

- Faible complexité de l'algorithme en termes de coût de traitement et d'échange de messages pour ordonnancer les paquets en ligne.
- Flexibilité de la politique pour traiter du trafic en plus (en excès) de celui garanti (lorsqu'une source émet à un rythme plus élevé que celui qu'elle négocié, il faut, de préférence, éviter le rejet systématique de paquets et privilégier la prise en compte des paquets non conformes au contrat avec une QoS dégradée).
- Passage à l'échelle (*scalability*) : la politique doit pouvoir fonctionner aussi pour des topologies avec un nombre élevé de nœuds de commutation.

6.2. Politiques d'ordonnancement

Pour servir les paquets de manière à respecter leurs exigences en QoS, différentes techniques d'ordonnancement de paquets ont été proposées. Nous nous limitons ici à présenter les politiques les plus répandues pour la garantie de délai.

6.2.1. Ordonnancement à priorité fixe

Des priorités sont affectées, de manière statique, aux trafics. Une file d'attente est allouée à chaque niveau de priorité et le paquet se trouvant en tête de la file la plus prioritaire est servi en premier, les autres devront attendre. La difficulté avec cette politique est de choisir correctement les priorités en fonction des contraintes de QoS à respecter.

6.2.2. Ordonnancement équitable ('Fair queuing')

Différentes formes de politiques dites équitables ont été proposées ; la plus connue étant WFQ ('Weighted fair queuing'). WFQ consiste à : 1) allouer une fraction de la bande passante à chaque flux (ou connexion), 2) calculer les instants d'émission des paquets de chaque flux en fonction de la fraction de bande passante allouée à ce flux et les paquets précédemment émis pour ce flux et 3) émettre en premier le paquet dont l'instant d'émission est le plus petit.

Il est prouvé que WFQ permet de garantir un débit et un délai de transfert bornés. C'est la raison pour laquelle WFQ est largement implémentée sur les routeurs offrant de la QoS. Par exemple, si on considère un flux i caractérisé par enveloppe de trafic $A_i(t) = b_i + r_i t$ (c'est-à-dire un trafic conforme à un seau percé (b_i, r_i)) qui traverse H routeurs fonctionnant tous avec WFQ, alors le délai de bout en bout est borné par (notez que ce délai est indépendant du nombre de flux concurrents) :

$$\frac{b_i}{R_i} + \frac{(H-1)M_i}{R_i} + \sum_{h=1}^{H} \frac{L^h_{\max}}{v^h}$$

où R_i ($R_i \geq r_i$) désigne le débit alloué au flux i, M_i la taille maximale des paquets du flux i, L^h_{max} la taille maximale de paquets traversant le routeur h, v_h le débit du lien de sortie de h.

Un des gros inconvénients de WFQ est la complexité du calcul des échéances pour la transmission des paquets. Pour remédier à cet inconvénient, plusieurs améliorations ont été proposées, a savoir SCFQ (Self Clocked Fair Queueing), SFQ (Start-time Fair Queueing), W^2FQ (Worst-case WFQ). Ces politiques sont présentées dans [7].

6.2.3. Ordonnancement EDF

C'est une forme d'ordonnancement à priorité dynamique où la priorité est affectée à chaque paquet au moment de son arrivée. A chaque flux est associé un délai maximal que le routeur doit respecter. A noter qu'un des problèmes avec cette discipline de service est la détermination du délai pour chaque flux (surtout si ce flux a un comportement non déterministe). L'échéance utilisée pour ordonnancer un paquet est égale à la somme de son instant d'arrivée et du délai affecté au flux auquel appartient ce paquet. L'ordonnanceur sélectionne toujours le paquet dont l'échéance est la plus petite. Il est démontré que si un flux i traverse un ensemble de routeurs implémentant tous une politique EDF, alors le délai de bout en bout est borné à condition que tous les trafics qui partagent des liens avec le flux i aient un modèle de trafic borné. En effet, dans la politique EDF, il n'y a pas d'isolation de flux au niveau des routeurs, par conséquent un flux qui envoie plus de trafic que ce qu'il a déclaré au moment de l'établissement de connexion peut remettre en cause le respect des échéances des autres flux. C'est la raison pour laquelle la politique EDF est souvent associée à *shapper* pour retarder les paquets en provenance de source qui émettent à un rythme plus élevé que celui demandé.

6.2.4. Partage de lien hiérarchique

Une des techniques implantées par les fabricants de routeurs à QoS est *Class Based Queueing* (CBQ) [12]. Le principe de CBQ est de coupler deux mécanismes : un ordonnanceur de partage de lien et un ordonnanceur général. L'ordonnanceur de partage de lien est chargé de veiller à ce que chaque classe de la structure hiérarchique reçoive bien sa part de bande passante, alors que l'ordonnanceur général détermine l'ordre d'émission des paquets sur le lien, tout en respectant les limitations imposées par le mécanisme de partage de lien.

Le partage de lien hiérarchique consiste à diviser la capacité d'un lien entre plusieurs sources organisées en *classes* dans une structure arborescente où le nœud racine est le lien lui-même, et où chaque nœud feuille correspond à une classe émettrice (une source). La propriété d'équité de CBQ réside dans : 1) chaque classe doit pouvoir recevoir au moins sa part théoriquement attribuée de bande passante, 2) la bande passante excédentaire doit pouvoir être utilisée et distribuée entre les différents flux qui en auraient besoin et 3) la bande passante excédentaire d'une classe devrait être mise à la disposition de ses classes soeurs en priorité.

Pour faire respecter ces principes, plusieurs types de directives ont été donnés par Floyd et Jacobson dans [12]. On dispose d'un *estimateur* qui permet de connaître l'état de chaque classe. A partir de cette connaissance, le *régulateur* déterminera quelles classes doivent être suspendues, et à quels moments. L'estimateur et le régulateur composent l'ordonnanceur de partage de lien.

L'ordonnanceur général est celui qui est chargé de déterminer l'ordre d'émission des paquets. Il est composé d'un ordonnanceur à priorités fixes et d'un algorithme de type PRR (*Packet Round Robin*) ou WRR (*Weighted Round Robin*), mais d'autres algorithmes peuvent éventuellement être utilisés.

La figure 3 montre un exemple d'utilisation de l'ordonnancement hiérarchique tel que le préconise certains travaux (standards) UMTS.

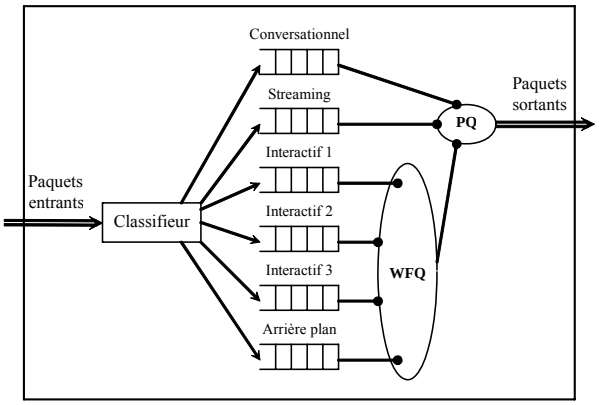


Figure 3. Ordonnancement hiérarchique dans UMTS.

6.3. Problèmes ouverts

Même si l'ordonnancement de paquets est l'un des sujets les plus traités depuis une quinzaine d'années, il reste encore beaucoup à faire pour aboutir à des architectures de réseau répondant aux différentes formes de trafics et d'exigences en termes de QoS. Il s'agit notamment des problèmes suivants :

- élaboration d'algorithmes (heuristiques) efficaces pour prendre en compte, simultanément, plusieurs contraintes de QoS ;
- détermination de bornes de délai (pour des flux agrégés, flux apériodiques…) ;
- adaptation dynamique des paramètres de l'ordonnancement aux caractéristiques des flux ;
- conception conjointe d'ordonnancement, routage, contrôle d'admission et réservation de ressources ;
- ordonnancement dans les réseaux sans fil ad hoc où les capacités des liens sont dynamiques (puisque les liens apparaissent et disparaissent en fonction des déplacements des stations) ;
- analyse des effets des mécanismes de sécurité sur les délais de transfert de bout en bout.

7. Routage

7.1. Routage à QoS

Dans le domaine du routage, il y a deux notions : le routage basé sur une politique (*policy-based routing)* et le routage basé sur des contraintes *(constraint-based routing)*. Le routage basé sur une politique signifie généralement que la prise de décision liée au choix des chemins est prise sur la base de politiques administratives (qui considèrent des configurations statiques pour des raisons de sécurité ou de coût, par exemple) et non sur la connaissance de la topologie du réseau et de métriques. Le routage basé sur des contraintes est plus large que le routage basé sur la QoS. Il répond à des contraintes multiples y compris des contraintes de QoS et des contraintes de politique. Dans la suite, nous nous intéressons uniquement au routage basé sur la QoS (ou simplement routage à QoS).

Le routage à QoS est défini comme : un mécanisme de routage avec lequel les chemins sont déterminés en fonction de la connaissance sur la disponibilité des ressources et des exigences en QoS des flux [8, 15, 16].

Comme les protocoles de routage actuellement utilisés dans Internet, tels que OSPF (*Open Shortest Path First*), RIP (*Routing Information Protocol*) et BGP (*Border Gateway Protocol*), ne prennent pas en compte les besoins de QoS, il a fallu définir des protocoles pour prendre en compte la QoS, c'est-à-dire des protocoles permettant le choix de chemins qui répondent aux exigences de QoS. Les principales considérations de conception d'algorithmes de routage à QoS sont liées aux aspects suivants :

- *Métriques et leur propagation* : le choix des métriques (bande passante, délai d'aller-retour, gigue…) est très important en fonction de la QoS à garantir. Se pose ensuite plusieurs questions aussi délicates et difficiles les unes que les autres : à quelles fréquences faut-il faire les mesures ? A quels points du réseau et au niveau de quelles couches effecteur les mesures ? Entre quels équipements les valeurs de métriques doivent-elles être échangées ? Quelle est la complexité des mécanismes de mesure ? Les réponses à ces questions sont très variées, ce qui montre la diversité et difficulté du problème de routage.
- *Sélection de chemin* : en utilisant les métriques collectées, on sélectionne un chemin en fonction des contraintes de QoS à satisfaire. Le problème n'est pas trivial. En effet, il est prouvé que le problème de recherche de chemin est NP-complet dès que l'on considère deux ou plusieurs types de contraintes de QoS non corrélées (par exemple, délai et bande passante, délai et disponibilité…) [22]. En général, on utilise des heuristiques pour calculer les chemins répondant à plusieurs critères de QoS. La sélection du chemin est étroitement

liée à la réservation de ressources. Lorsqu'un chemin est trouvé, il faut confirmer la réservation de ressources sur ce chemin.

- *Passage à l'échelle* : les informations liées aux métriques peuvent être nombreuses et conduire à un surcoût inacceptable pour leur gestion. Pour assurer le passage à l'échelle, ces informations doivent être agrégées et les échanges de chaque information doivent être limités aux seuls points où cette information est utilisée.
- *Imprécision des informations de routage* : les flux sont souvent aléatoires ; certaines éléments du réseau peuvent aussi se connecter ou se déconnecter selon des règles dépendantes ou indépendantes des flux. La conséquence est qu'il est difficile d'avoir des informations de métriques qui soient précises et qu'il faut choisir des chemins en tenant compte de cette imprécision. Une question se pose alors : quelle est la crédibilité des valeurs de métriques sans remettre en cause trop souvent les décisions de routage ? La plupart des algorithmes de routage sont conçus pour tenir compte de cette imprécision.

7.2. Routage intra-domaine et inter-domaine

Le chemin emprunté par un flux peut traverser un seul domaine (un réseau d'entreprise par exemple) géré par une seule autorité ou traverser plusieurs domaines gérés par des autorités indépendantes. Il y a donc deux types de routage : routage intra-domaine et routage inter-domaine. Le routage intra-domaine se charge de trouver des chemins à l'intérieur d'un même domaine en fonction de politiques ou règles locales à ce domaine et d'informations de routage maintenues à l'intérieur de ce domaine. Le routage inter-domaine quant à lui a pour objectif de trouver des chemins sur plusieurs domaines en tenant compte des politiques ou règles de ces domaines qui ne sont pas les mêmes (et qui peuvent même être antagonistes pour des raisons de sécurité par exemple). Les différences liées aux politiques et informations sur les métriques rendent le problème de routage inter-domaine très complexe pour respecter la QoS de bout en bout. Parfois ces différences peuvent être telles qu'il est impossible de garantir une certaine QoS. Par exemple, considérons trois domaines, le premier garantit seulement le débit, le second garantit seulement le taux de perte et le troisième garantit seulement la disponibilité. Un flux qui traverse les trois domaines ne peut être sûr qu'aucune de ses contraintes de débit, de taux de perte ou de disponibilité ne peut être garantie.

7.3. Algorithmes de routage à QoS

7.3.1. Types de routage selon la prise de décision

En prenant en compte la manière dont les informations de routage sont gérées et les décisions de sélection de chemin sont prises, les algorithmes de routage sont habituellement regroupés en trois classes : par la source, distribué (ou saut par saut) et hiérarchique.

Dans le routage par la source, chaque nœud a une connaissance globale sur l'état du réseau et il sélectionne le chemin à emprunter en fonction de cette connaissance et de la destination des paquets. Une fois le chemin sélectionné, le nœud signale aux autres qu'un chemin a été sélectionné et qu'ils doivent réserver les ressources adéquates. Ce type de routage est simple et ne conduit pas à des boucles. Malheureusement, il présente des problèmes de passage à l'échelle, car chaque nœud est obligé de gérer toute l'information d'état du réseau.

Dans le routage saut-par-saut, chaque nœud connaît juste le prochain nœud (saut) pour atteindre la destination. Ainsi, quand un paquet arrive dans le nœud, il est expédié vers le prochain saut et ainsi de suite jusqu'à ce que le paquet arrive à destination. C'est le type de routage le plus utilisé actuellement dans Internet et c'est le plus pratique pour s'adapter aux protocoles de routage existants. L'inconvénient est que les nœuds (indépendants) peuvent avoir des vues inconsistantes les unes avec les autres ce qui peut conduire des paquets à boucler dans le réseau. Il peut présenter aussi des problèmes de passage à l'échelle.

Le routage hiérarchique est surtout utile pour les grands réseaux. Le réseau est structuré en plusieurs niveaux hiérarchiques ; les niveaux les plus bas (feuilles) correspondent aux nœuds physiques (i.e. les routeurs ou commutateurs). Ces nœuds sont organisés en groupes, les groupes forment le deuxième niveau hiérarchique, ensuite on constitue des groupes de groupes, etc. Les informations de routage sont gérées par les nœuds de bordure de chaque groupe. Chaque nœud représentant un groupe connaît les informations sur les autres groupes. A chaque niveau hiérarchique, on peut avoir un algorithme de routage adapté. Le principal avantage de ce type de routage est sa capacité de passage à l'échelle. Son inconvénient est qu'avec les niveaux hiérarchiques, les informations d'état sont agrégées ce qui conduit à une perte de précision sur l'état réel des nœuds physiques.

7.3.2. Types de routage selon le nombre de récepteurs

Lorsqu'on considère le nombre de récepteurs de paquets, on distingue trois types de routage : routage *unicast*, routage *multicast* et routage *anycast*.

Dans le routage unicast, un seul émetteur et un seul récepteur sont concernés par les échanges. C'est le type de routage le plus implémenté et pour lequel on sait, dans beaucoup de cas, garantir la QoS.

Des applications telles que la vidéoconférence, les espaces de travail partagés et la simulation distribuée interactive font appel au routage multicast, car il y a presque toujours plusieurs récepteurs pour un même flux. Les récepteurs d'un flux forment un groupe. On peut avoir des groupes denses ou épars selon leur proximité géographique, des groupes ouverts ou fermés

selon les règles d'intégration du groupe, des groupes permanents ou temporaires selon leur durée de vie, des groupes statiques ou dynamiques selon que leurs membres sont fixes ou apparaissent et disparaissent dynamiquement… Le routage multicast est plus difficile à gérer et pose de nombreuses questions notamment les suivantes :

- Comment choisir et maintenir les arbres multicast ? Comment centrer l'arbre multicast autour de la source d'information pour minimiser les communications ?
- Comment réserver et utiliser de manière efficace des ressources homogènes ou hétérogènes selon les besoins et capacités des membres du groupe ?
- Avec quelles ressources minimales peut-on faire fonctionner un groupe puisque les membres peuvent arriver de manière dynamique ? Comment accepter ou refuser des membres uniquement pour des raisons de QoS ? Quels membres sont primordiaux dans un groupe et avec quelle QoS ?
- Quels sont les effets de la gestion de groupe (entrée et sortie des membres) sur le bon fonctionnement de la QoS ?
- Comment agréger des flux multicast sans trop de perte de précision en QoS obtenue par les flux ?

La communication anycast est très utile dans des contextes tels que les serveurs Web miroirs. Le routage anycast consiste en : étant donné la requête d'un client (pour un service donné), sélectionner parmi les serveurs possibles (qui offrent ce service) celui dont le chemin client-serveur permet de respecter les contraintes de QoS du client. Ce mode de routage pose de nombreux problèmes, notamment :

- Critères de sélection du serveur : choisir n'importe lequel pourvu que la QoS soit garantie ? le premier trouvé ? tenir compte de la charge des serveurs ? celui qui donne la meilleure QoS ?
- Mécanismes de signalisation : doit-on passer par un nœud qui connaît les serveurs potentiels ? ou doit-on rechercher directement les serveurs ? Comment gérer la sélection des serveurs si ces derniers sont répartis sur différents domaines ?
- Y a-t-il une coordination entre les serveurs pour équilibrer leur charge et mieux garantir la QoS ?

7.4. Routage à QoS dans les réseaux sans fil

La dynamique des réseaux sans fil est beaucoup plus forte que celle des réseaux filaires ce qui rend la tâche de garantie de QoS beaucoup plus difficile. En effet, avec des équipements qui bougent fréquemment, et donc des liens qui se créent et qui disparaissent rapidement, il est difficile d'avoir des informations d'état à jour. Ainsi, la durée de vie des chemins est plus courte et qu'il faut sans cesse rechercher de nouveaux chemins. Depuis quelques années déjà, beaucoup de travaux ont été effectués pour proposer des algorithmes de routage adaptés au monde sans fil. Une des solutions pour gérer le routage est de distinguer les liens stationnaires et les liens transitoires. Les liens stationnaires sont ceux qui existent entre des nœuds fixes ou qui bougent peu et qui ont par conséquent une durée de vie plus longue. Les liens transitoires quant à eux concernent les nœuds qui se déplacent (rapidement ou souvent). Il faut donc privilégier les liens stationnaires pour sélectionner les chemins. Malheureusement, les liens stationnaires n'ont pas toujours suffisamment de ressources pour répondre aux besoins de QoS et il faut donc compter avec les liens transitoires pour sélectionner les chemins.

7.5. Problèmes de routage ouverts

Même si de nombreux travaux ont été faits sur le routage, il demeure un sujet complexe, et des nombreux problèmes restent ouverts, pour diverses raisons :

- Les applications ont des besoins très variés en termes de QoS, or on sait que le routage avec deux ou plusieurs contraintes indépendantes est un problème NP-complet.
- Les chemins empruntés par les flux peuvent traverser des domaines avec des politiques de gestion de QoS différentes (voire antagonistes).
- La diversité des protocoles de routage rend difficile l'interopérabilité entre domaines tout en garantissant des chemins avec une QoS de bout en bout.
- La plupart des réseaux souhaitent véhiculer des paquets avec différents niveaux de service tout en acceptant du trafic de type meilleur effort. A quel niveau faut-il dégrader la QoS de certains flux pour contenter tout le monde ?
- La charge du réseau change parfois de manière brutale et aléatoire à cause de l'arrivée des flux mais aussi de la dynamique de la topologie du réseau (notamment les réseaux sans fil).

8. Réservation de ressources et négociation

8.1. Réservation de ressources

Les échanges liés à un flux nécessitent l'utilisation de ressources (mémoire, unité centrale et bande passante). Pour fournir et maintenir des niveaux de QoS, la gestion des ressources doit être dirigée par les besoins en QoS.

Lorsqu'un chemin possédant suffisamment de ressources est trouvé pour répondre aux besoins de QoS d'un flux (individuel ou agrégé), les ressources adéquates sont réservées.

Il y a deux approches de réservation de ressources : réservation statique et réservation dynamique.

La réservation statique consiste à n'accepter un flux que si toutes les ressources dont il a besoin sont disponibles et une fois le flux accepté, les ressources restent réservées (et non disponibles pour les autres flux) jusqu'à la fin de ce flux. Cette approche a le mérite de la simplicité, mais aussi de la clarté pour les utilisateurs (ou on les accepte ou on les rejette et s'ils sont acceptés, ils verront les besoins de QoS respectés). Malheureusement, cette approche conduit souvent à de mauvaises performances globales du réseau. Il s'agit d'une approche non adaptative.

La réservation dynamique consiste à allouer les ressources au fur et à mesure qu'elles sont réellement demandées par les paquets. Les ressources ne sont pas réservées explicitement pour un flux, mais pour plusieurs flux qu'ils utilisent de manière banalisées. Dans les techniques de réservation dynamique, on a celles qui sont adaptatives et celles qui ne le sont pas. Par techniques adaptatives, on entend des techniques ou le réseau peut dégrader la QoS fournie à certains flux pour s'adapter aux surcharges, mais aussi des techniques où les applications s'adaptent en fonction de l'état du réseau (elles sont moins exigeantes) quand le réseau est surchargé. L'adaptation du réseau et/ou des applications c'est l'idéal, mais c'est plus facile à dire qu'à faire. En effet, les techniques adaptatives nécessitent plus de complexité au niveau des fonctions.

8.2. Négociation et renégociation

Avant d'allouer les ressources à un flux, l'ensemble des composants (au niveau de toutes les couches concernées) doivent s'assurer que collectivement ils peuvent répondre aux besoins de QoS du flux à accepter. Pour garantir le respect de la QoS promise, l'ensemble du réseau doit contrôler l'usage des ressources et éventuellement réallouer des ressources en cas de besoin.

Les mécanismes de négociation sont utilisés pour établir un niveau de QoS qui soit réalisable par le réseau et acceptable par les utilisateurs. Cela nécessite la participation d'au moins trois éléments : la source de données, le réseau et le destinataire des données. Dans le cas d'une communication de groupe (une ou plusieurs sources et un ou plusieurs destinataires), plusieurs participants sont concernés et la négociation peut s'avérer plus complexe que dans le cas d'une communication de type 'unicast'. Chaque participant dans la communication informe les autres de son acceptation ou refus des flux en fonction de ses capacités et des spécificités des flux à accepter. Un participant qui refuse un flux peut proposer des valeurs de paramètres qu'il peut respecter et les soumet aux autres : c'est la *négociation*. Selon les applications, les paramètres de QoS peuvent être négociés en bloc, un par un, pour chaque paquet, pour toute la durée d'une connexion, à l'abonnement de l'utilisateur, etc.

L'opération de *renégociation* de QoS est très importante dans les environnements où la charge varie de manière significative dans le temps. Elle permet à des utilisateurs de revoir (à la baisse ou à la hausse) leurs exigences en termes de QoS. Une des difficultés de mise en place de la renégociation est le choix des instants où cette renégociation peut être déclenchée. En général, ces instants peuvent correspondre aux instants où l'utilisateur demande explicitement de changer de valeurs de paramètres de QoS, aux instants où le contrôle et supervision de l'état du réseau indiquent des changements importants au niveau de la charge et aux instants d'interruption ou de défaillance de certains composants de communication.

Beaucoup de choses restent à faire ou à améliorer concernant la réservation de ressources, nomment les aspects suivants :

- réservation et utilisation de ressources pour les flux agrégés ;
- techniques adéquates de réservation de ressources dans les réseaux mobiles ;
- spécification et signalisation des possibilités d'adaptation des applications.

9. Gestion de buffers, contrôle de congestion

On ne peut gérer (et ordonnancer) convenablement les paquets transitant par chaque nœud que si ce nœud dispose de suffisamment de mémoire pour stocker les paquets entrants. Par conséquent, la gestion de mémoire (dite aussi gestion de buffers) joue un rôle important dans la garantie de QoS. Comme le comportement de certaines sources n'est pas connu a priori, il se peut que certains nœuds du réseau se trouvent en situation de congestion. Les paquets arrivant pendant les phases de congestion peuvent saturer des nœuds et conduire à des rejets pour manque de mémoire.

Pour qu'un réseau puisse fournir des garanties de QoS, il doit intégrer des capacités robustes de contrôle de congestion lui permettant de détecter les surcharges de trafic et d'éliminer des paquets. Plusieurs méthodes sont utilisées pour prendre en compte le problème de congestion (qui est un des problèmes fondamentaux des réseaux) : méthodes réactives et approches préventives.

9.1. Méthodes réactives

En cas de détection de seuil de congestion, les nœuds doivent prendre des décisions quant aux paquets à rejeter. Faut-il rejeter : des paquets au hasard ? Les derniers paquets arrivant ? Des paquets issus de sources qui acceptent un taux de perte élevé ? Un ou plusieurs paquets par source de manière équitable ? Les paquets les moins prioritaires ? Les paquets qui demandent un service best effort ? Les paquets marqués par leur source comme pouvant être rejetés ? Le choix entre ces possibilités est un compromis entre efficacité en termes de QoS fournie et complexité de la gestion des buffers.

9.2. Méthodes préventives

Parce qu'il est souvent impossible de prévoir suffisamment de ressources mémoire pour absorber toutes les situations de surcharge dues à une accélération de la transmission par certaines sources, il faut prévoir des mécanismes de rejet de paquets. Ainsi, au lieu d'attendre que les situations de congestion apparaissent pour les traiter, beaucoup (presque tous) les concepteurs de réseaux préfèrent des méthodes préventives en mettant en place des techniques d'évitement de congestion. Parmi ces techniques, les plus répandues sont EPD (Early Packet Discard), RED (Random Early Discard) et ECN (Explicit Congestion Notification).

Dans la technique EDP, un seuil d'utilisation des buffers est défini. Quand ce seuil est atteint, les paquets entrants sont rejetés sans distinction de leur origine (ce qui n'est pas toujours une bonne décision). Des améliorations de EDP ont été proposées pour plus d'équité au niveau rejet en fonction des sources des paquets. Malheureusement, l'équité se paye par une complexité accrue de la technique de rejet.

RED est la technique la plus connue pour la prévention de congestion. C'est une technique de gestion de queue dite active [11]. Elle a été introduite pour améliorer les performances de TCP. RED fonctionne selon un principe probabiliste de rejet de paquets entrants. Au niveau de chaque nœud, RED utilise deux seuils *S_bas* et *S_haut*. RED estime la taille moyenne de la file d'attente, *Qest* et compare la valeur estimée aux seuils *S_bas* et *S_haut*. Si *Qest* < *S_bas* alors RED accepte le paquet entrant. Si *Qest* > *S_haut* alors RED élimine le paquet entrant. Si *S_bas* ≤ *Qest* ≤ *S_bas S_haut,* alors RED élimine les paquets entrants avec une probabilité *P*(*Qest)* qui est fonction de *Qest*.

Des extensions ont été proposées pour améliorer RED : Weighted RED, BLUE, RIO (RED with Input-Output bit), FRED (Fair RED), SRED (Stabilized RED).

ECN est une autre approche permettant de mieux gérer les congestions dans les réseaux IP [20]. Au lieu de rejeter les paquets en cas de dépassement de seuil de congestion, ECN avertit les systèmes hôtes en positionnant des bits dans les paquets. C'est une technique qui demande la participation des hôtes pour réduire les rejets tout en évitant les congestions. Si les hôtes ne réagissent pas au marquage de paquets par les routeurs, cette technique devient inefficace. ECN peut s'utiliser conjointement avec RED, WRED…

Le problème de gestion des buffers reste encore ouvert et demande des travaux concernant notamment les questions suivantes :

- Comment mettre en place des mécanismes de gestion de buffers adaptifs en fonction des applications (selon qu'elles acceptent les pertes ou non, selon qu'elles sont à l'écoute ou non du réseau quant aux notifications de congestion…) ?
- Comment combiner de manière efficace les mécanismes de rejet avec l'ordonnancement de paquets ? Souvent on commence par l'acceptation ou le rejet d'un paquet entrant, ensuite quand le paquet entrant est accepté, la fonction d'ordonnancement est lancée pour savoir à quel moment ce paquet peut être servi. Cette approche peut être inefficace puisqu'on ne prend pas (ou peu) en compte les contraintes de temps du paquet dans le mécanisme de rejet actuellement utilisés.

10. Mapping de QoS

Les besoins en QoS s'expriment de différentes manières selon les services offerts par le support d'exécution aux applications. Prenons par exemple le cas d'une application qui manipule des films. Cette application peut exprimer un besoin de QoS en termes de qualité de son et d'image et de synchronisation entre les deux flux (si elle sait que le système sous-jacent peut répondre à ce besoin) ou bien elle peut exprimer un besoin en termes de débit binaire et elle gère par elle-même le reste des fonctions nécessaires pour présenter un film. Pour passer de la QoS exprimée à un niveau à un autre niveau, des fonctions de traduction/translation sont nécessaires (on parle de "mapping de QoS" ou de dérivation de QoS aussi).

Beaucoup de composants (hôtes, routeurs, systèmes d'exploitation…) coopèrent pour offrir une QoS de bout en bout. Dans le réseau, chaque couche, selon les protocoles qu'elle utilise, peut offrir sa propre forme de QoS. Par ailleurs, différentes formes de QoS peuvent être fournies par les composants (hétérogènes) traversés. Il est donc important de pouvoir translater ('mapper') [10] :

- les paramètres de QoS d'une couche à une autre, (par exemple de la couche Transport à la couche Réseau),
- d'un domaine de réseau à un autre (par exemple, entre deux domaines DiffServ ou entre un domaine DiffServ et un domaine IntServ),
- d'un type de réseau à un autre (par exemple, entre un réseau IP et un réseau ATM),
- d'un système d'exploitation à un autre,
- d'une plateforme à une autre,
- d'un langage à un autre,
- …

Les objectives du mapping de QoS sont de pouvoir combiner les formes de QoS fournies par différents composants (vues à différents niveaux de granularité) pour fournir de la QoS de bout en bout. Les travaux faits actuellement sur le mapping de QoS concernent certains des aspects énumérés précédemment, par exemple, déterminer le débit à partir de la période et taille des

paquets pour un réseau IP. Il manque encore des approches rigoureuses et complètes pour réaliser le mapping des paramètres de QoS. L'idéal serait de disposer de boîtes à outils qui déterminent et valident le mapping de manière automatique, cela permet d'une part de valider plus facilement les applications et les réseaux et d'autre part de permettre plus de portabilité des applications ayant des contraintes de QoS. Il reste donc du travail à effectuer.

11. Autres fonctions de gestion de QoS

Nous présentons ici brièvement des fonctions liées au contrôle et supervision de la QoS. Même si ces fonctions sont présentées de manière succincte (faute de place), elles n'en demeurent pas moins importantes que celles présentées précédemment. Elles sont vitales pour la gestion de QoS dynamique, et elles sont aussi parfois relativement complexes à mettre en œuvre et à analyser.

11.1. Contrôle de conformité de trafic

Un utilisateur peut, volontairement ou non, dépasser le rythme de transmission qu'il a demandé au moment de l'acceptation de son contrat. Une telle situation, si elle n'est pas détectée par le réseau, peut engendrer des perturbations conduisant éventuellement à ne plus fournir la QoS à certains flux même si ces derniers se comportent correctement. Pour éviter de tels inconvénients, le réseau met en place un mécanisme de contrôle des paquets. Les paquets non conformes au contrat de trafic sont soit détruits, soit marqués pour être servis à un faible niveau de priorité. Les techniques les plus utilisées pour le contrôle de la conformité du trafic sont celles du seau percé et seau à jetons.

11.2. Façonnage de trafic

Les paramètres de spécification de trafic peuvent être tels qu'il est impossible de décrire avec précision la taille et durée des rafales. La conséquence est qu'une source peut soumettre des paquets à des rythmes irréguliers même si la source est considérée comme périodique. Par ailleurs, comme les flux en provenance de différentes sources partagent des composants du réseau, les délais de transfert d'un flux peuvent varier à cause d'autres flux. En particulier, même si les paquets d'un flux périodique entrent dans le réseau de manière régulière (périodique), ils n'arrivent pas de manière régulière dans tous les nœuds qu'ils traversent. Pour essayer de garder le rythme d'arrivée des paquets d'un flux approximativement identique à celui de leur entrée dans le réseau, certains (ou tous les) nœuds intermédiaires retardent certains paquets ; on dit qu'il y a façonnage de flux ("traffic shaping"). Le façonnage de trafic permet de garantir une gigue bornée et réduit les situations de rafales de paquets. Le problème est que le façonnage utilisé partout ralentit le réseau et nécessite que tous les nœuds connaissent les spécifications de tous les trafics ce qui rend ce mécanise inefficace (voire inutilisable) dans les grands réseaux. En général, le façonnage est utilisé à l'entrée du réseau.

11.3. Supervision de QoS

Pendant la phase d'établissement de la QoS demandée par l'utilisateur, le réseau configure les mécanismes adéquats (de gestion de ressources, de routage, d'ordonnancement de paquets, etc.) pour offrir la QoS demandée. A l'exception de cas simples (notamment de gestion statique de QoS déterministe), l'acceptation d'un flux pendant la phase d'établissement de QoS ne signifie pas que tout est réglé et n'il y a plus rien à contrôler dynamiquement. En effet, des situations diverses (surcharges momentanées, pannes d'équipements, etc.) peuvent apparaître et conduire éventuellement à la détérioration de la QoS de certains flux. Pour cela, la supervision de QoS doit être mise en œuvre; elle consiste principalement à : 1) mesurer (à l'aide de différents indicateurs situés à différents nœuds et couches) la QoS réellement fournie aux utilisateurs (cette fonction est dite *Fonction de surveillance de QoS* ou parfois *Métrologie* comme expliqué dans le paragraphe suivant) et 2) à comparer la QoS mesurée par rapport à celle promise et, selon les capacités d'adaptation du réseau, alerter l'utilisateur (cette fonction est dite *Fonction d'alerte de QoS*), reconfigurer certains mécanismes de gestion de la QoS, par exemple, changer la priorité des paquets ou augmenter la taille de certains buffers (cette fonction est dite *Fonction de maintenance QoS*) ou faire les deux.

11.4. Métrologie de QoS

Normalement, pour chaque métrique (délai, débit, taux d'erreur…), on doit pouvoir mesurer la QoS fournie et la comparer à celle promise. Pour cela, il faut des techniques de mesure des différents indicateurs de QoS. Se posent alors des questions : Que faut-il mesurer ? Où faut-il mesurer ? Quand mesurer ? Comment mesurer ? Les réponses à ces questions sont nombreuses ; elles dépendent du réseau et de sa gestion et de l'utilisation des mesures pour agir sur l'allocation des ressources et l'adaptation de QoS.

La mesure devient une activité à part entière surtout avec le développement d'Internet, on parle de *métrologie* [19].

De manière simplifiée, il y a deux grandes approches de mesure : actives et passives. Dans les méthodes de mesure actives, on injecte du trafic pour mesurer certains éléments (par exemple, envoyer des paquets de sonde pour mesurer le délai d'aller-retour). Le principe des mesures passives consiste à regarder le trafic et d'étudier ses propriétés en un ou plusieurs points du réseau. L'avantage des mesures passives et qu'elles ne sont pas intrusives et ne changent rien à l'état du réseau. De plus, elles permettent des analyses très avancées.

Des problèmes se posent avec les deux types d'approches. Pour les approches actives : comment être sûr que le trafic de sonde subit le même traitement que le trafic normal pour extrapoler les résultats ? A quel rythme faut-il injecter le trafic de sonde pour qu'il soit à la fois représentatif et qu'il n'affecte pas les trafics normaux ? Pour les approches passives : il est très difficile de déterminer le service qui pourra être offert à un flux en fonction des informations obtenues en métrologie passive et il est difficile de prédire le comportement futur à partir de celui observé pendant un certain laps de temps.

Différents travaux actuels s'intéressent à l'exploitation en temps réel des mesures et à la corrélation entre les indicateurs de mesures (statistiques) pour agir sur la QOS.

11.5. Classification et marquage de paquets

Dans certains protocoles, comme DiffServ ou MPLS, pour allouer les ressources selon les besoins de QoS, un ensemble de classes de trafic est fixé a priori en fonction des niveaux de services que peut offrir le réseau. En général, on associe une file d'attente à chaque classe de trafic au niveau de chaque nœud du réseau. Les paquets entrant dans chaque nœud doivent être identifiés et affectés à une file d'attente en fonction de la QoS promise au flux auquel ces paquets appartiennent : c'est la *classification* de paquets. Dans le contexte d'Internet, la classification utilise les entêtes de paquets IP et des entêtes des protocoles supérieurs (par exemple, le numéro de port TCP) pour déterminer la file d'attente à utiliser. Certains protocoles de QoS, comme DiffServ et MPLS, marquent chaque paquet entrant (en modifiant un champ approprié dans le paquet) et cette marque détermine la classe de service avec laquelle sera traité le paquet dans le nœud où il a été marqué ou dans tout le réseau. La classification et marquage de paquet sont des opérations invoquées pour chaque paquet, leur implémentation doit se faire de manière à les rendre les plus rapides possible pour ne pas freiner le fonctionnement du réseau. Par ailleurs, plus il y a de marques (ou codes) différentes, plus on différencie les flux, le revers de la médaille étant la complexité de la solution pour gérer un nombre élevé de codes, donc de niveaux de QoS.

11.6. Signalisation de QoS

Pour que le réseau puisse répondre aux besoins de QoS, les systèmes hôtes doivent signaler au réseau (donc à tous les nœuds intermédiaires entre les sources et les destinations des flux) les caractéristiques de leurs trafics et les exigences en QoS. Cette signalisation peut se faire pendant la phase d'établissement de connexion ou d'autres phases (par exemple, à la négociation d'un contrat avec le réseau).

Un des protocoles les plus connus aujourd'hui pour la signalisation de QoS est RSVP ("Resource ReSerVation Protocol"), adapté aux communications *unicast* et *multicast*. RSVP permet de spécifier aux nœuds se trouvant sur le chemin emprunté par un flux les ressources à réserver et permet aussi des échanges périodiques ou non pour maintenir les réservations tant que le flux n'est pas fini. Malheureusement, RSVP pose des problèmes de passage à l'échelle. Différents travaux ont été consacrés aux extensions de RSVP pour les réseaux de grande taille (comme RSVP pour DiffServ) et les réseaux ad hoc pour que les informations d'état tiennent compte de la dynamique de ces réseaux.

11.7. Politique de QoS

Dans les fonctions de gestion de QoS décrites précédemment il y a souvent beaucoup de choix et d'alternatives possibles. Il faut donc une politique pour guider ces fonctions. Une politique est un ensemble de buts ou d'actions pour guider et maintenir les décisions en termes de satisfaction de besoins de QoS. Les politiques sont souvent exprimées en termes de règles pour administrer, gérer et contrôler l'accès au réseau. Elles définissent les critères importants, le poids de chaque critère, les décisions qui améliorent les critères importants pour la gestion du réseau vu comme une ressource globale… La politique de QoS définit aussi les objectifs globaux (économiques, stratégiques, de sécurité...).

12. Mobilité et QoS

Il y a de plus en plus d'équipements mobiles (PDA et autres) utilisés pour accéder à la messagerie électronique, envoyer des fax, téléphoner, écouter de la musique, regarder des films… pour des utilisateurs qui voyagent ou qui bougent. Par ailleurs, il y a de plus en plus d'applications dite sensibles à la position ("location-aware applications") dans lesquelles, les positions des utilisateurs (clients) sont déterminées par des moyens de localisation (tels le GPS) pour leur envoyer des infos diverses (météo, adresses des hôtels, cinémas et hôpitaux à proximité). De telles applications peuvent aussi être utilisées dans des situations critiques (par les secours…).

La mobilité est l'aptitude, pour un utilisateur, de continuer à utiliser les services auxquels il a souscrit malgré ses déplacements. Pour que la mobilité soit complètement transparente, les réseaux doivent répondre aux besoins de QoS sous des conditions de déplacements éventuellement fréquents et rapides des terminaux.

Pour pouvoir assurer une certaine continuité de la QoS lors des déplacements, il faudrait des mécanismes de localisation, de propagation des positions et de prédiction des futures positions des terminaux. Différents travaux sont menés dans ce sens depuis quelques années et soutenus notamment par l'IETF (pour IP mobile), 3GPP et 3GPP2.

Les situations de dégradation (voire de la rupture) de QoS se rencontrent essentiellement quand un mobile change de cellule (opération de *handoff*) en restant dans le même réseau ou quand il change de réseau. Un des thèmes de recherche récent sur la mobilité est celui l'approximation de la position future en utilisant des modèles de comportement des mobiles (homme d'affaire, employé de bureau, véhicules roulant en milieu urbain…). Les modèles de position de mobiles font appel à beaucoup de notions notamment la géométrie, les statistiques, le comportement humain… A ce titre, ils constituent un domaine de recherche à part entière dans le monde des réseaux et Télécoms [6].

L'approximation de la position future joue un rôle important dans l'anticipation pour la réservation des ressources le long du trajet supposé du mobile pour ne pas rompre ou dégrader la QoS qui lui est offerte.

Un autre aspect est celui de la non dégradation de la QoS en cas de changement de réseau d'origine du mobile (le *roaming*). Cela nécessite des négociations entre domaines pour réserver les ressources.

La mobilité 'aléatoire' des usagers rend difficile la modélisation en termes de phénomènes stochastiques des flux temps réel qui traversent une cellule ou un réseau pour allouer efficacement les ressources.

La problématique de la QoS dans les réseaux mobiles se situe donc principalement au niveau de la gestion de ressources pour des utilisateurs qui bougent. Il y a, à la fois, des travaux sur des modèles de réservation de ressources à l'avance et sur les protocoles de réservations (par exemple, il y a eu des extensions de RSVP qui ont été proposées pour les réseaux ad hoc).

Des techniques comme "Localized Predictive Resource reservation" [23] ont été proposées pour la gestion de ressources dans les réseaux IP sans fil.

3GPP et 3GPP2 ont adopté DiffServ pour garantir la QoS pour leurs réseaux IP. IntServ a été jugé peu adapté (voire inadapté) à l'esprit de la mobilité ; il rendrait les choses trop complexes à gérer.

Dans les environnements adaptatifs, les utilisateurs négocient des contrats (SLA) avec le réseau. Ensuite, pour minimiser le coût de gestion de la mobilité, chaque mobile n'a pas besoin de signaler de manière individuelle les demandes de ressources sur le chemin sur lequel il se déplace. Il a cependant besoin de méthodes pour la négociation de QoS et l'adaptation aux offres du réseau. Un mobile peut négocier à la baisse la QoS au lieu de se déconnecter à chaque fois que la QoS initialement promise n'est plus garantie suite à ses déplacements.

Deux grandes tendances sont actuellement explorées pour la réservation de ressources : 1) on connaît le chemin par lequel va passer le mobile et on anticipe les réservations, cela revient à peu près au même problème que celui de la réservation dans un réseau fixe ou 2) on ne connaît pas le chemin, mais on essaie (avec un certain horizon d'anticipation) estimer par où va passer le mobile et réserver les ressources. Cette deuxième approche est la plus efficace, mais la plus complexe à définir, analyser et mettre en œuvre surtout pour certains paramètres de QoS comme le délai ou la gigue.

D'autres problèmes plus difficiles à surmonter sont dus au fait qu'un mobile peut passer dans des zones qui perturbent (ou bloquent complètement) la communication (un tunnel par exemple). Ceci nécessite des approches différentes permettant de s'accommoder de situations difficiles de la communication. Accepter une rupture momentanée de la communication, puis reprendre la communication dès que possible devient alors une nécessité d'adaptation pour certains mobiles.

Un autre aspect est celui de la consommation d'énergie. En effet, les équipements mobiles sont munis de batteries dont l'usage doit se faire avec précaution. Un équipement qui reste à l'écoute ou qui transmet puise dans sa batterie. Comment, par exemple, continuer à téléphoner ou regarder un film, suivre une vidéoconférence en mode dégradé permettant de ne pas puiser trop dans la batterie ? Comment participer à une communication ou offrir une partie de ses capacités pour servir de relais (lors d'une sollicitation pour trouver un chemin), sans compromettre sa batterie ? sont des questions propres aux environnements mobiles qui méritent des réponses appropriées. Les techniques de routage et de réservation de ressources proposées pour les réseaux filaires ne sont pas directement utilisables dans le cas des réseaux de mobiles non filaires.

Etant donnés les enjeux économiques et stratégiques, la prédiction de localisation de mobiles et fourniture de QoS dans les réseaux mobiles constituent aujourd'hui des thèmes de recherche très actifs où les propositions abondent ce qui rend leur analyse difficile.

13. Conclusion

La notion de QoS concerne différents aspects, notamment le temps de réponse, la fiabilité et la disponibilité du service fourni aux utilisateurs. Il existe actuellement de nombreuses solutions (touchant à différents aspects : algorithmes, protocoles, mécanismes, architectures…) permettant la fourniture de QoS. Chaque type de solution est généralement adapté à un contexte particulier et présente des performances qui lui sont spécifiques. Les situations sont trop nombreuses pour avoir une (ou un nombre réduit de) solution(s) adaptée(s) à tous les cas de figures. Il faut souvent trouver un compromis entre la complexité de la solution et les performances souhaitées pour répondre aux exigences des applications.

Le déploiement de mécanismes de gestion de QoS reste un sujet complexe et nécessite beaucoup de connaissances sur les différentes facettes liées aux fonctions de gestion de QoS comme nous l'avons

souligné dans les sections précédentes. Aussi, la problématique de la QoS est loin d'être maîtrisée et la multiplicité des travaux de recherche en cours en témoigne. Nous avons souligné quelques sujets (challenges) qui nous semblent importants et qui peuvent susciter des discussions aussi bien en considérant chaque fonction de QoS séparément qu'en composant différents mécanismes pour aboutir à une architecture de QoS. Notre présentation s'est focalisée sur un des acteurs de la QoS : le réseau. Il est important de souligner que les autres acteurs (l'utilisateur, l'application, le middleware et le système d'exploitation) jouent aussi des rôles dans la QoS ; leur(s) vue(s) de la QoS doit (doivent) être considérée(s) pour obtenir réellement de la QoS utile.

Références

[1] I.O. Aagedal, Quality of Service Support in Development of Distributed Systems, PhD thesis, University of Oslo, 2001.

[2] C. Aurrecoechea, A.T. Campbell, L. Hauw, "A Survey of QoS Architectures", *ACM Multimedia Systems Journal*, Vol. 6, pp. 138-151, 1998.

[3] S. Blake et al. An Architecture for Differentiated Service, RFC 2475, 1998.

[4] L. Breslau et al, "Endpoint Admission Control: Architectural Issues and Performance", *ACM SIGCOMM*, 2000, pp. 57-69.

[5] L. Breslau, S. Jamin, S. Shenken, "Comments on the Performance of Measurement-Based Admission Control Algorithms", Proc. of IEEE INFOCOM 2000.

[6] C. Cheng, R. Jain, E. van den Berg. Location Prediction Algorithms for Mobile Wireless Systems. In M. Illyas and B. Furht, editors, *Handbook of Wireless Internet*. CRC Press, 2003.

[7] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri. *Scheduling in Real-Time Systems*, John Wiley, 2002.

[8] E. Crawley et al., A Framework for QoS-based Routing in the Internet, RFC 2386, 1998.

[9] B. Chatterjee, M. Sydir, T. Lawrence, "Taxonomy of QoS Specifications", in Proceedings of *WORDS'97*, Newport Beach, California, February 1997.

[10] L.A. DaSilva, "QoS Mapping along the Protocol Stack: Discussion and Preliminary Results", in Proceedings of *IEEE Int. Conference on Communications (ICC'00)*, June 18-22, 2000, New Orleans, LA, pp. 713-717.

[11] S. Floyd, V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Vol. 1, pp. 397-413, 1993.

[12] S. Floyd, V. Jacobson, "Link-sharing and Resource Management Models for Packet Networks", *IEEE/ACM Transactions on Networking*, Vol. 3, pp. 365-386, 1995.

[13] S. Frølund, J. Koistinen, "Quality-of-Service Specification, in Distributed Object Systems", *Distributed Systems Engineering Journal*, Vol. 5, pp. 179-202, 1998.

[14] R. Guérin, H. Ahmadi, M. Naghshineh, "Equivalent Capacity and its Applications to Bandwidth Allocation in High-Speed Networks", *Journal of Selected Areas in Communications*, Vol. 9, pp. 968-981, 1991.

[15] R. Guerin, A. Orda, "QoS Routing in Networks with Inaccurate Information: Theory and Algorithms", *IEEE/ACM Transactions on Networking*, Vol. 7, pp. 350-364, 1999.

[16] R. Guérin, V. Peris, "Quality of Service in Packet Networks – Basic Mechanisms and Directions", *Computer Networks*, Vol. 31, pp. 169-179, 1999.

[17] ISO/IEC CD 13236-2, Open Distributed Processing - Quality of service framework - Basic framework, July 1995.

[18] Z. Mammeri, "Framework for Parameter Mapping to Provide End-to-end QoS Guarantees in IntServ/DiffServ architectures", *Computer Communications*, Vol. 28, pp. 1074-1092, 2005.

[19] V. Paxson et al., Framework for IP Performance Metrics, RFC 2330, 1998

[20] K. Ramakrishnan, S. Floyd, A Proposal to Add Explicit Congestion Notification (ECN) to IP, RFC 2481, 1999.

[21] S. Shenker, J. Wroclawski, Network Element Service Specification Template, RFC 2216, 1997.

[22] Z. Wang, J. Crowcroft, "Quality of Service Routing for Supporting Multimedia Applications", *IEEE Journal on Selected Areas in Communications*, Vol. 14, pp. 1288-1234, 1996.

[23] T. Zhang et al., "Local Predictive Reservation for Handoff in Multimedia Wireless IP Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 19, pp. 1931-1941, 2001.

Control Task Timing and Quality of Control

Anton Cervin, Dan Henriksson, Bo Lincoln, Martin Andersson, Karl-Erik Årzén
Department of Automatic Control, LTH
Lund University
Sweden
anton@control.lth.se

Abstract

Most embedded control systems have constrained computational resources. This makes codesign of the control algorithms and the real-time scheduling an interesting issue. There is also a strong trend in the real-time community of moving away from the hard real-time model. Hence, embedded controllers must be designed so that they are aware of the resource constraints and timing uncertainty of the implementation platform. This paper discusses temporal determinism, design approaches, and codesign tools for such systems. In particular, the codesign tools Jitterbug and TrueTime are described.

1 Introduction

The current pervasive and ubiquitous computing trend has increased the emphasis on embedded and networked computing within the engineering community. Today embedded computers already by far outnumber desktop computers. Embedded systems are often found in consumer products and are therefore subject to hard economic constraints. Some examples are automotive systems and mobile phones. The pervasive nature of these systems generates further constraints on physical size and power consumption. These product-level constraints give rise to resource constraints on the computing platform level, for example, limitations on computing speed, memory size, and communication bandwidth. Due to economic considerations, this is true in spite of the fast development of computing hardware. In many cases, it is not economically justified to add an additional CPU or to use a processor with more capacity than what is required by the application. Cost also favors general-purpose computing components over specially designed hardware and software solutions.

Control systems constitute an important subclass of embedded computing systems—so important that, for example, within automotive systems, computers commonly go under the name electronic control units (ECUs). A top-level modern car contains more than 50 ECUs of varying complexity. Most of these ECUs implement different feedback control tasks, for instance, engine control, traction control, anti-lock braking, active stability control, cruise control, and climate control.

At the same time, control systems are becoming increasingly complex from both the control and computer science perspectives. Today, even seemingly simple embedded control systems often contain a multi-tasking real-time kernel and support networking. Many computer-controlled systems are distributed, consisting of computer nodes and a communication network connecting the various systems. It is not uncommon for the sensor, actuator, and control calculations to reside on different nodes. This gives rise to networked control loops. Within individual nodes, controllers are often implemented as one or several tasks on a microprocessor with a real-time operating system. Often, the microprocessor also contains tasks for other functions, such as communication and user interfaces. The operating system typically uses multiprogramming to multiplex the execution of the various tasks. The CPU time and the communication bandwidth can hence be viewed as shared resources for which the tasks compete.

By tradition, the design of computer-based control systems is based on the principle of *separation of concerns*. This separation is based on the assumption that feedback controllers can be modeled and implemented as periodic tasks that have a fixed period, T, a known worst-case bound on the execution time (WCET), C, and a *hard deadline*, D. The latter implies that it is imperative that the tasks always meet their deadlines, i.e., that the actual execution time (response time) is always less or equal to the deadline, for each invocation of the task. This is in contrast to a *soft deadline*, which may occasionally be violated. The fixed-period assumption of the simple task model has also been widely adopted by the control community and has resulted in the development of the sampled computer-control theory with its assumption of deterministic, equidistant sampling. The separation of concerns has allowed the control community to focus on the pure control design without having to worry about how the control system eventually is implemented. At the same time, it has allowed the real-time computing community to focus on development of scheduling theory and computa-

tional models that guarantee that hard deadlines are met, without any need to understand what impact scheduling has on the stability and performance of the plant under control.

Historically, the separated development of control and scheduling theories for computer-based control systems has produced many useful results and served its purpose well. However, the separation has also had negative effects. The two communities have become partly alienated. This has led to a lack of mutual understanding between the fields. The assumptions of the simple model are also overly restrictive with respect to the characteristics of many control loops. Many control loops are not periodic, or they may switch between a number of different fixed sampling periods. Control loop deadlines are not always hard. On the contrary, many controllers are quite robust to variations in sampling period and response time. Hence, it is questionable whether it is necessary to model them as hard-deadline tasks.

The main drawbacks of the separation of concerns are that it does not always utilize the available computing resources in an optimal way, and that it sometimes gives rise to worse control performance than what can be achieved if the designs of the control and real-time computing parts are integrated. This is particularly important for embedded control applications with limited computing and communication resources, demanding performance specifications, and high requirements on flexibility. For these types of applications, better performance can be achieved if a codesign approach is adopted, where the control system is designed taking the resource constraints into account and where the real-time computing and scheduling is designed with the control performance in mind. The resulting *implementation-aware control systems* are better suited to meet the requirements of embedded and networked applications.

2 The Codesign Problem

The *control and scheduling codesign problem* can in the uniprocessor case be informally stated as follows:

> *Given a set of processes to be controlled and a computer with limited computational resources, design a set of controllers and schedule them as real-time tasks such that the overall control performance is optimized.*

An alternative view of the same problem is to say that we should design and schedule a set of controllers such that the least expensive implementation platform can be used while still meeting the performance specifications. For distributed systems, the scheduling is extended to also include the network communication.

The nature and the degree of difficulty of the codesign problem for a given system depend on a number of factors:

- *The real-time operating system.* What scheduling algorithms are supported? How is I/O handled? Can the real-time kernel measure task execution times and detect execution overruns and missed deadlines?

- *The scheduling algorithm.* Is it time driven or event driven, priority driven or deadline driven? What analytical results regarding schedulability and response times are available? What scheduling parameters can be changed on-line? How are task overruns handled?

- *The controller synthesis method.* What design criteria are used? Are the controllers designed in the continuous-time domain and then discretized or is direct discrete design used? Are the controllers designed to be robust against timing variations? Should they actively compensate for timing variations?

- *The execution-time characteristics of the control algorithms.* Do the algorithms have predictable worst-case execution times? Are there large variations in execution time from sample to sample? Do the controllers switch between different internal modes with different execution-time profiles?

- *Off-line or on-line optimization.* What information is available for the off-line design and how accurate is it? What can be measured on-line? Should the system be able to handle the arrival of new tasks? Should the system be re-optimized when the workload changes? Should there be feedback from the control performance to the scheduling algorithm?

- *The network communication.* Which type of network protocol is used? Can the protocol provide worst-case guarantees on the network latency? How large is the probability of lost packets?

Codesign of control and computing systems is not a new topic. Control applications were one of the major driving forces in the early development of computers. At that time, limited computer resources was a general problem, not only a problem for embedded controllers. For example, the issues of limited word length and fixed-point calculations and their results on resolution were well known among control engineers in the 1970s. However, as computing power has increased, these issues have received decreasing attention. A nice survey of the area from the mid-1980s is given in [15].

3 Temporal Determinism

Computer-based control theory normally assumes equidistant sampling and negligible, or constant, input-output latencies. However, this situation can seldom be achieved in practice or is too costly for a particular application. In a multi-threaded system, tasks interfere with each other due to preemption and blocking from task communication. Execution times may be data dependent or vary due to the use of caches. In networked control loops,

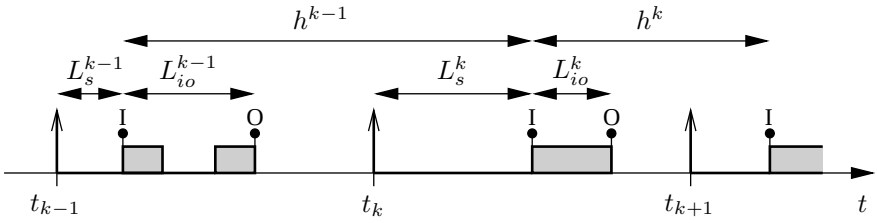


Figure 1. Controller timing

where the sensors, controllers, and actuators reside on different physical nodes, the communication gives rise to latencies that can be more or less deterministic, depending on the network protocols used. The result of all this is jitter in sampling intervals and non-negligible and varying latencies.

The basic timing parameters of a control task are shown in Fig. 1. It is assumed that the control task is *released* (i.e., inserted into the ready queue of the real-time operating system) periodically at times given by $t_k = hk$, where h is the *nominal sampling interval* of the controller. Due to preemption and blocking from other tasks in the system, the actual *start* of the task may be delayed for some time L_s. This is called the *sampling latency* of the controller. A dynamic scheduling policy will introduce variations in this interval. The *sampling jitter* is quantified by the difference between the maximum and minimum sampling latencies in all task instances,

$$J_s \stackrel{\text{def}}{=} \max_k L_s^k - \min_k L_s^k. \tag{1}$$

Normally, it can be assumed that the minimum sampling latency of a task is zero, in which case we have $J_s = \max_k L_s^k$. Jitter in the sampling latency will also introduce jitter in the sampling interval h. From the figure, it is seen that the actual sampling interval in period k is given by

$$h^k = h - L_s^{k-1} + L_s^k. \tag{2}$$

The *sampling interval jitter* is quantified by

$$J_h \stackrel{\text{def}}{=} \max_k h^k - \min_k h^k. \tag{3}$$

We can see that the sampling interval jitter is upper bounded by

$$J_h \leq 2J_s. \tag{4}$$

After some computation time and possibly further preemption from other tasks, the controller will actuate the control signal. The delay from the sampling to the actuation is called the *input-output latency*, denoted L_{io}. Varying execution times or task scheduling will introduce variations in this interval. The *input-output jitter* is quantified by

$$J_{io} \stackrel{\text{def}}{=} \max_k L_{io}^k - \min_k L_{io}^k. \tag{5}$$

It is well known that a constant input-output latency decreases the phase margin of the control system, and that it introduces a fundamental limitation on the achievable closed-loop performance. The resulting sampled-data system is time invariant and of finite order, which allows standard linear time-invariant (LTI) analysis to be used (see, e.g., [5]). For a given value of the latency, it is easy to predict the performance degradation due to the delay. Furthermore, it is straightforward to account for a constant latency in most control design methods. From this perspective, a constant input-output latency is preferable over a varying latency.

The scheduling-induced input-output latency of a single control task can be reduced by assigning it a higher priority (or, alternatively, under deadline-based scheduling, a shorter deadline). Of course, this approach will not work for the whole task set.

Another option is to use non-preemptive scheduling. This will guarantee that, once the task has started its execution, it will continue uninterrupted until the end. The disadvantages of this approach are that the scheduling analysis for non-preemptive scheduling is quite complicated (e.g., [20, 28]), and that the schedulability of other tasks may be compromised. However, as computing speed increases, and, hence, $C \ll T$, it becomes increasingly interesting to execute the tasks with hard deadlines non-preemptively.

A standard way to achieve a short input-output latency in a control task is to separate the algorithm calculations in two parts: *Calculate Output* and *Update State*. Calculate Output contains only the parts of the algorithm that make use of the current sample information. Update State contains the update of the controller states and precalculations for the next period. Update State can therefore be executed after the output signal transmission, hence, reducing the input-output latency. Further improvements can be obtained by scheduling the two parts as subtasks with different priorities, see [8].

A control system with a time-varying input-output latency is quite difficult to analyze, since the standard tools for LTI systems cannot be used. If the statistical properties of the latency are known, then theory from jump linear systems can be used to evaluate the stability and performance of the system (in the mean sense), see [26].

4 Design Approaches

The temporal non-determinism caused by the implementation platform can be approached in two different

ways:

- the hard real-time approach, or
- the soft, control-based approach.

The hard real-time approach strives to maximize the temporal determinism by using special-purpose hardware, software, and protocols. This includes techniques such as static cyclic scheduling, time-triggered computing and communication [21], synchronous programming languages [6], and computing models such as Giotto [18]. This approach has several advantages, especially for safety-critical applications. For example, it simplifies attempts at formal verification. The approach also has drawbacks. It has strong requirements on the availability of realistic worst-case bounds on resource utilization, something which is very difficult to obtain in practice. A result of this could be underutilization and, possibly, poor control performance due to sampling intervals that are too long. The approach also makes it difficult to use general-purpose implementation platforms. This is particularly serious, since it is these systems that have the most advantageous price-performance development.

The soft, control-based approach instead views the temporal non-determinism caused by the implementation platform as an uncertainty or disturbance acting on the control loop and handles it using control-based approaches. This can be done using a number of techniques. The simplest way is to rely on the inherent robustness of feedback. It is well known that feedback increases the robustness towards plant variations. The same holds for variations caused by the implementation platform, i.e., *temporal robustness*. Another approach to deal with jitter in the control design is to explicitly design the controller to be robust, i.e., treat the delay as a parametric uncertainty. Many robust design methods are available, such as H_∞, quantitative feedback theory (QFT), and μ-design. The majority of these methods are developed for plant uncertainties. Although parts of the results carry over to temporal robustness, it is likely that there is room for much more research here.

It is also possible to let the controller actively compensate for the delay in each sample. This can be compared to traditional gain-scheduling and feedforward from disturbances. An optimal, jitter-compensating controller was developed in [26]. The controller compensates for time-varying delays in a control loop, which is closed over a communication network. The setup is shown in Fig. 2. The sensor node samples the process periodically, sending the measurements over the network to the controller node. The controller node is event driven and computes a new control signal as soon as a measurement arrives. The control signal is sent to the event-driven actuator node, which outputs the signal to the process. The linear-quadratic (LQ) state feedback control law has the form

$$u(k) = -L(\tau_{sc}^k) \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix}, \qquad (6)$$

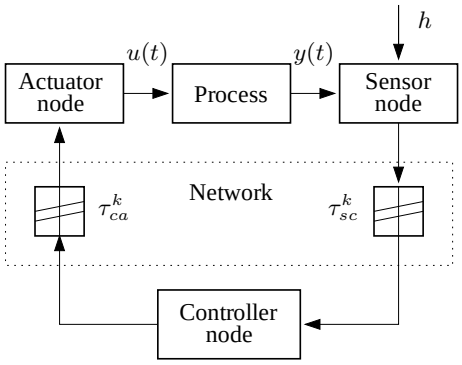


Figure 2. Distributed digital control system with network communication delays τ_{sc}^k and τ_{ca}^k. From [26].

where the feedback gain L depends on the sensor-to-controller delay τ_{sc}^k in the current sample. The computation of the gain vector L is quite involved and requires that the probability distributions of τ_{sc} and τ_{ca} be known. The state feedback can be combined with an optimal state observer that takes the actual delays into account.

The above approach cannot be directly applied to scheduling-induced delays. The problem is that the delay in the current sample will not be known until the task has finished, and by then it is too late to compensate. A simple scheme that compensates for delay in the previous sample is presented in [22]. The compensator has the same basic structure as the well-known Smith predictor, but allows for a time-varying delay.

Many other heuristic jitter compensation schemes have been suggested; see e.g., [14, 3, 25]. What the approaches have in common is that they require language or operating system support for instrumenting an application with measurement code.

In order to fully apply these techniques, it is necessary to increase the understanding of how temporal non-determinism affects control performance. This requires new theories and tools that are now beginning to emerge. An important issue that is still lacking is a theory that allows us to determine which level of temporal determinism a given control loop really requires in order to meet given control objectives on stability and performance. Is it necessary to use a time-triggered approach, or will an event-based approach perform satisfactorily? How large are the input-output latencies that can be tolerated? Is it OK to now and then skip a sample in order to maintain the schedulability of the task set? Ideally, one would like to have an index that decides the required level of temporal determinism through a single quantitative measure. One possible name for such an index would be the schedulability margin. This measure would need to combine both a margin with respect to input-output latency and jitter and a margin that decides how large a sampling jitter the loop can tolerate. For constant input-output latencies the classical phase margin can be applied.

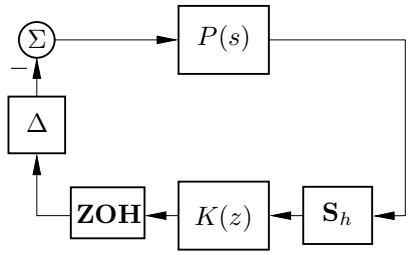


Figure 3. Computer-controlled system with continuous-time plant $P(s)$, periodic sampler $\mathbf{S}_h$, discrete-time controller $K(z)$, zero-order hold, and time-varying delay Δ.

An extension of the classical delay margin to time-varying delays is proposed in [11]. The jitter margin $J_m(L)$ is defined as the largest input-output jitter for which closed-loop stability is guaranteed for any time-varying latency $\Delta \in [L, L + J_m(L)]$, where L is the constant part of the input-output latency. The jitter margin is based on the following stability theorem, presented in [19].

Theorem 1 (Stability under output jitter) *Consider the computer-controlled system in Figure 3, where the plant is described by the linear continuous-time system $P(s)$, and the plant output is sampled with the constant interval h. The controller is described by the linear discrete-time system $K(z)$. Following the zero-order hold, there is a time-varying delay Δ before the control signal is applied to the input of the plant. The system is stable for any time-varying delays $\Delta \in [0,\, Nh]$, where $N > 0$ is a real number, if*

$$\left| \frac{P_{\rm alias}(\omega)K(e^{i\omega})}{1 + P_{\rm ZOH}(e^{i\omega})K(e^{i\omega})} \right| < \frac{1}{\tilde{N}\left|e^{i\omega} - 1\right|}, \quad \forall \omega \in [0,\, \pi], \tag{7}$$

where $\tilde{N} = \sqrt{\lfloor N \rfloor^2 + 2\lfloor N \rfloor g + g}$ and $g = N - \lfloor N \rfloor$; $P_{\rm ZOH}(z)$ is the zero-order hold discretization of $P(s)$, and

$$P_{\rm alias}(\omega) = \sqrt{\sum_{k=-\infty}^{\infty} \left| P\left(i(\omega + 2\pi k)\frac{1}{h} \right) \right|^2}. \tag{8}$$

Proof: *See [19].*

The jitter margin can be used to derive hard deadlines that guarantee closed-loop stability, provided that the scheduling method employed can provide bounds on the worst-case and best-case response times of the controller tasks. What is still missing in order to be able to define a reasonable analytical concept for a schedulability margin is a simple sampling jitter criterion. The criterion should ideally tell the size of the variations around a nominal sampling interval that the process can tolerate, maintaining stability and acceptable performance.

In addition to being temporally robust, it is also important for a control system to be robust towards faults. Numerous theories and methods have been developed for fault detection, diagnosis, and fault tolerance within the control community. However, the majority of this work concerns faults that occur within the plant, sensors, or actuators. As most software engineers are sadly aware, faults in the software system are far more common than faults in the plant under control. In spite of this, the amount of work that considers robustness against these types of faults, i.e., *functional robustness*, is very small. In [13] a method is presented that renders a control system more robust to computer-level faults leading to data errors. The method is based on the introduction of artificial signal limits in combination with an anti-windup scheme. Related to this, in [4], a methodology is developed for analyzing the impact that these types of data errors have on control system dependability.

In order for codesign of control and computing systems to become feasible, it is necessary to have software tools that allow the designers to analyze and simulate how timing affects control performance. Such tools have recently begun to emerge, e.g., [12, 27, 24]. In the following sections, two such tools will be described: Jitterbug (http://www.control.lth.se/~lincoln/jitterbug) and TrueTime (http://www.control.lth.se/~dan/truetime).

5 Jitterbug

Jitterbug [9, 23, 10] is a MATLAB-based toolbox that computes a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The main contribution of the toolbox, which is built on well-known theory (linear quadratic Gaussian (LQG) theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problem.

Jitterbug offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous-time and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a random delay (described by a discrete probability density function) can be specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The

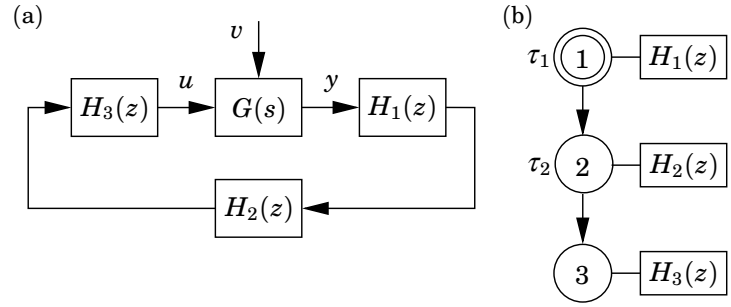


Figure 4. A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the three discrete-time systems $H_1(z)$, $H_2(z)$, and $H_3(z)$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.

signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a Jitterbug model is shown in Fig. 4, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

5.1 Signal Model

A *continuous-time system* is described by

$$\begin{aligned} \dot{x}_c(t) &= Ax_c(t) + Bu(t) + v_c(t) \\ y(t) &= Cx_c(t), \end{aligned}$$

where A, B, and C are constant matrices, and v_c is a continuous-time white noise process with covariance R_{1c}. (In the toolbox, it is also possible to specify discrete-time measurement noise. This will be interpreted as input noise at any connected discrete-time system.) The cost of the system is specified as

$$J_c = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{bmatrix} x_c(t) \\ u(t) \end{bmatrix}^T Q_c \begin{bmatrix} x_c(t) \\ u(t) \end{bmatrix} dt,$$

where Q_c is a positive semidefinite matrix.

A *discrete-time system* is described by

$$\begin{aligned} x_d(t_{k+1}) &= \Phi x_d(t_k) + \Gamma u(t_k) + v_d(t_k) \\ y(t_k) &= Cx_d(t_k) + Du(t_k) + e_d(t_k), \end{aligned}$$

where Φ, Γ, C, and D are possibly time-varying matrices (see below). The covariance of the discrete-time white noise processes v_d and e_d is given by

$$R_d = E \begin{bmatrix} v_d(t_k) \\ e_d(t_k) \end{bmatrix} \begin{bmatrix} v_d(t_k) \\ e_d(t_k) \end{bmatrix}^T.$$

The input signal u is sampled when the system is updated, and the state x_d and the output signal y are held between updates. The cost of the system is specified as

$$J_d = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{bmatrix} x_d(t) \\ u(t) \end{bmatrix}^T Q_d \begin{bmatrix} x_d(t) \\ u(t) \end{bmatrix} dt,$$

where Q_d is a positive semidefinite matrix. Note that the update instants t_k need not be equidistant in time, and that the cost is defined in continuous time.

The *total system* is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. Throughout, MIMO formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous- and discrete-time systems:

$$J = \sum J_c + \sum J_d.$$

5.2 Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can also be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers and also greatly simplifies the cost calculations.

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = \begin{bmatrix} P_\tau(0) & P_\tau(1) & P_\tau(2) & \dots \end{bmatrix},$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain δ is a constant that is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period.

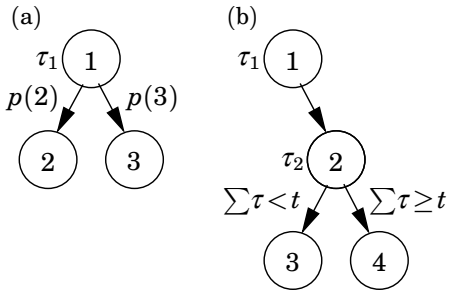


Figure 5. Alternative execution paths in a Jitterbug execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

Node- and Time-Dependent Execution

The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers for example.

Alternative Execution Paths

For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In Jitterbug, two such cases can be modeled (see Fig. 5):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a timevector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

5.3 Computation of Cost and Spectral Densities

The computation of the total cost is performed in three steps: First, the cost functions, the continuous-time noise, and the continuous-time systems are sampled using the time-grain of the model. Second, the closed-loop system is formulated as a jump linear system, where Markov nodes are used to represent the time-steps in and between the execution nodes. Third, the stationary variance of all states in the system is calculated.

For periodic systems, the Markov state always returns to the periodic execution node every h/δ time steps. The stationary variance in the periodic execution node can then be obtained by solving a linear system of equations. The cost is then calculated over the time steps in one period. In this case, the cost calculation is fast and exact. It is also straightforward to compute the spectral densities of all outputs as observed in the periodic timing node. For systems without a periodic node, the variance must be computed iteratively. In both cases, the toolbox will return an infinite cost if the total system is not stable (in the mean-square sense). More details about the internal workings of Jitterbug can be found in [10].

5.4 Example: Networked Control System

As an example, we will study a networked control system such as the one in Figure 2. We will begin by investigating how sensitive the control loop is to slow sampling and delays, and then we will look at delay and jitter compensation.

The Jitterbug model of the system was shown in Fig. 4. The DC servo process is given by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}.$$

The process is driven by white continuous-time input noise. There is assumed to be no measurement noise.

The process is sampled periodically with the interval h. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

and the discrete-time PD controller is implemented as

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right),$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$. (A real implementation would include a low-pass filter in the derivative part, but that is ignored here.)

The delays in the computer system are modeled by the two (possibly random) variables τ_1 and τ_2. The total delay

```
G = 1000/(s*(s+1));                          Define the process
H1 = 1;                                      Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                    Define the controller
H3 = 1;                                      Define the actuator

Ptau1 = [ ... ];          Define delay probability distribution 1
Ptau2 = [ ... ];          Define delay probability distribution 2

N = initjitterbug(delta,h);                  Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);              Define timing node 1
N = addtimingnode(N,2,Ptau2,3);              Define timing node 2
N = addtimingnode(N,3);                      Define timing node 3

N = addcontsys(N,1,G,4,Q,R1,R2);  Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);                  Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);                  Add controller to node 2
N = adddiscsys(N,4,H3,3,3);                  Add actuator to node 3

N = calcdynamics(N);                         Calculate internal dynamics
J = calccost(N);                             Calculate the total cost
```

Figure 6. This MATLAB script shows the commands needed to compute the performance index of the networked control system using Jitterbug.

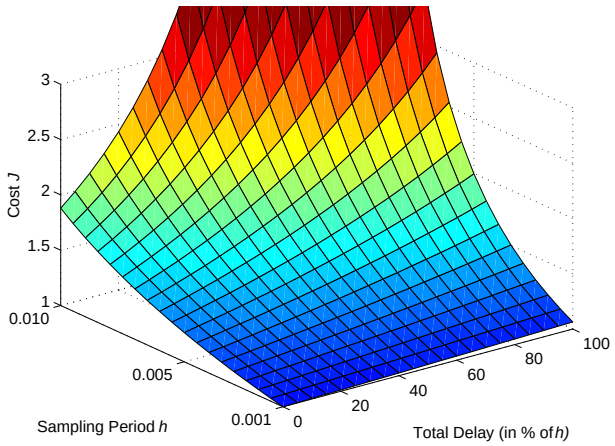


Figure 7. Example of a cost function computed using Jitterbug. The plot shows the cost as a function of sampling period and delay in the networked control system example.

from sampling to actuation is thus given by $\tau_{tot} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise Jitterbug would skip the remaining updates).

Finally, we need to specify the control performance criterion to be evaluated. As a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(y^2(t) + u^2(t) \right) dt. \qquad (9)$$

An outline of the MATLAB commands needed to specify the model and compute the value of the cost function are given in Fig. 6.

Sampling Period and Constant Delay

A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control, rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that

$$0.2 < \omega_b h < 0.6,$$

where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of computational delay is typically not considered in such rules of thumb, however. Using Jitterbug, the combined effect of sampling period and computational delay can be easily investigated. In Fig. 7, the cost function (9) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant total delay ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation

If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at run-time will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since Jitterbug allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the Jitterbug model of the networked control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $\tau_{max}/2$, where τ_{max} denotes the maximum round-trip delay in the loop. A range of PD controller parameters (ranging from $K = 1.5$ and $T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $\tau_{max}/4$.

The sampling interval is set to $h = 10$ ms to make the effects of delay and jitter clearly visible. In Fig. 8, the cost function (9) has been evaluated with and with-

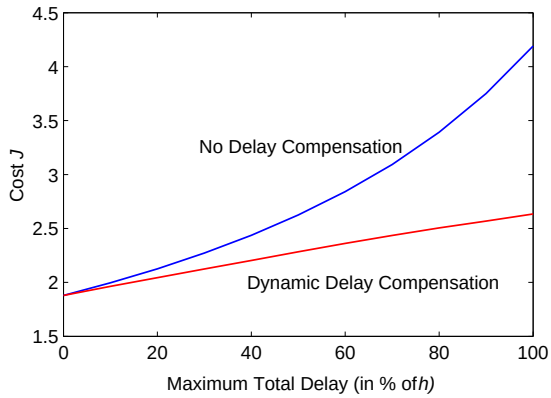


Figure 8. Cost as a function of maximum delay in the networked control system.

out delay compensation for values of the maximum delay ranging from 0 to 100% of the sampling interval. The cost increases much more rapidly for the uncompensated system. The same example will be studied in more detail later using the TrueTime simulator.

6 TrueTime

TrueTime [9, 16, 17] is a MATLAB/Simulink-based tool that facilitates simulation of the temporal behavior of a multi-tasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary continuous-time Simulink blocks. The TrueTime toolbox introduce new Simulink blocks modeling real-time kernels and computer networks. The blocks are event-driven and execute user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual computer blocks is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model.

The level of simulation detail can also be chosen by the user—it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TrueTime allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data-dependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

TrueTime can be used in several ways:

- to investigate the effects of timing nondeterminism, caused, for example., by preemption or transmission delays, on control performance
- to develop compensation schemes that adjust the controller dynamically based on measurements of actual timing variations
- to experiment with new, more flexible approaches to dynamic scheduling, such as feedback scheduling of CPU time and communication bandwidth and quality-of-service (QoS)-based scheduling approaches
- to simulate event-driven control systems (e.g., engine controllers and distributed controllers).
- to simulate energy-constrained wireless networked embedded systems such as sensor/actuator networks and mobile robots.

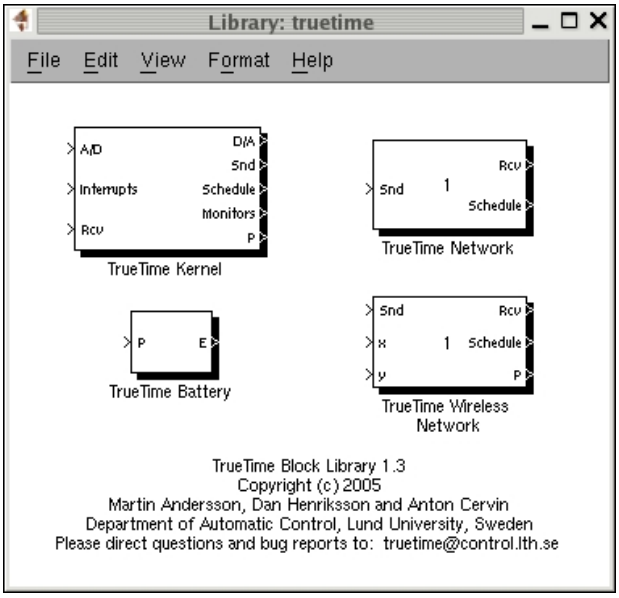


Figure 9. The TrueTime 1.3 block library.

6.1 Simulation Environment

The interfaces to the computer and network Simulink blocks are shown in Fig. 9. The blocks are event-driven, with the execution determined both by internal and external events. Internal events are timely and correspond to events such as “a timer has expired,” “a task has finished its execution,” or “a message has completed its transmission.” External events correspond to external interrupts, such as “a message arrived on the network” or “the crank angle passed zero degrees.”

The block inputs are assumed to be discrete-time signals, except the signals connected to the A/D converters of the computer block, which may be continuous-time signals. All outputs are discrete-time signals. The Schedule and Monitors outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

The blocks are realized as variable-step, discrete, MATLAB S-functions written in C++, the Simulink engine being used only for timing and interfacing with the rest of the model (the continuous dynamics). It should thus be easy to port the blocks to other simulation environments, provided these environments support event detection (zero-crossing detection).

6.2 The Computer Block

The computer block S-function simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.

Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors and timers that have been created for the simulation.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Tasks

The task is the main construct in the TrueTime simulation environment. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers.

An arbitrary number of tasks can be created to run in the TrueTime kernel. Each task is defined by a set of attributes and a code function. The attributes include a name, a release time, a worst-case execution time, an execution time budget, relative and absolute deadlines, a priority (if fixed-priority scheduling is used), and a period (if the task is periodic). Some of the attributes, such as the release time and the absolute deadline, are constantly updated by the kernel during simulation. Other attributes, such as period and priority, are normally kept constant but can be changed by calls to kernel primitives when the task is executing.

Furthermore, it is possible to associate three different interrupt handlers with each task. A task termination handler will be triggered when the code function of the task has executed its last segment, see below. A default termination handler is provided by the kernel for periodic tasks. This simply updates the release and absolute deadline and puts the task to sleep until next period. Similar to Real-Time Java [7], two overrun handlers may also be attached to each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution time overrun handler (triggered if the task executes longer than its worst-case execution time).

Interrupts and Interrupt Handlers

Interrupts may be generated in two ways: externally or internally. An external interrupt is associated with one of the external interrupt channels of the computer block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used to simulate engine controllers that are sampled against the rotation of the motor or distributed controllers that execute when measurements arrive on the network.

Internal interrupts are associated with timers. Both periodic timers and one-shot timers can be created. The corresponding interrupt is triggered when the timer expires. Timers are also used internally by the kernel to implement the overrun handlers described in the previous section.

When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt. An interrupt handler works much the same way as a task, but is scheduled on a higher priority level. Interrupt handlers will normally perform small, less time-consuming tasks, such as generating an event or triggering the execution of a task. An interrupt handler is defined by a name, a priority, and a code function. External interrupts also have a latency during which they are insensitive to new invocations.

Priorities and Scheduling

Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or nonpreemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns a deadline implies deadline-driven scheduling. Predefined priority functions exist for most of the commonly used scheduling schemes.

Code

The code associated with tasks and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code is normally divided into several segments, as shown in Fig. 10. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output delays, blocking when accessing shared resources, etc. The simulated execution time of each segment is returned by the code function, and can be modeled as constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption from higher-priority activities and interrupts may cause the actual delay between the segments to be longer than the execution time.

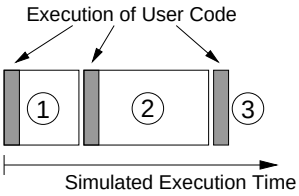


Figure 10. The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times.

```
function [exectime, data] = myController(seg, data)
switch seg,
   case 1,
      data.y = ttAnalogIn(1);
      data.u = calculateOutput(data.y);
      exectime = 0.002;
   case 2,
      ttAnalogOut(1, data.u);
      updateState(data.y);
      exectime = 0.003;
   case 3,
      exectime = -1; % finished
end
```

Figure 11. Example of a simple code function.

Fig. 11 shows an example of a code function corresponding to the time line in Fig. 10. The function implements a simple controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution, which will trigger execution of the termination handler of the task.

The functions `calculateOutput` and `updateState` are assumed to represent the implementation of an arbitrary controller. The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Besides A/D and D/A conversion, many other kernel primitives exist that can be called from the code functions. These include functions to send and receive messages over the network, create and remove timers, perform monitor operations, and change task attributes. Some of the kernel primitives are listed in Table 1.

Graphical Controller Representation

As an alternative to textual implementation of the controller algorithms, TrueTime also allows for graphical representation of the controllers. Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions, using the primitive `ttCallBlockSystem`. A block diagram of a PI controller is shown in Fig. 12. The block system has two inputs, the reference signal and the process output, and two outputs, the control signal and the execution time.

Table 1. Examples of kernel primitives (pseudo syntax) that can be called from code functions associated with tasks and interrupt handlers.

`ttAnalogIn(ch)`	Get the value of an input channel
`ttAnalogOut(ch, val)`	Set the value of an output channel
`ttSendMsg(rec,data,len)`	Send message over network
`ttGetMsg()`	Get message from network input queue
`ttSleepUntil(time)`	Wait until a specific time
`ttCurrentTime()`	Current time in simulation
`ttCreateTimer(time,ih)`	Trigger interrupt handler at a specific time
`ttEnterMonitor(mon)`	Enter a monitor
`ttWait(ev)`	Await an event
`ttNotifyAll(ev)`	Activate all tasks waiting for an event
`ttSetPriority(val)`	Change the priority of a task
`ttSetPeriod(val)`	Change the period of a task

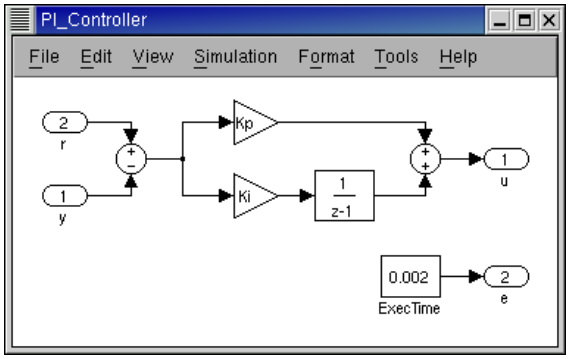


Figure 12. Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. This example shows a PI controller.

Synchronization

Synchronization between tasks is supported by monitors and events. Monitors are used to guarantee mutual exclusion when accessing common data. Events can be associated with monitors to represent condition variables. Events may also be free (i.e., not associated with a monitor). This feature can be used to obtain synchronization between tasks where no conditions on shared data are involved. The example in Fig. 13 shows the use of a free event `input_event` to simulate an event-driven controller task. The corresponding `ttNotifyAll`-call of the event is typically performed in an interrupt handler associated with an external interrupt port.

Output Graphs

Depending on the simulation, several different output graphs are generated by the TrueTime blocks. Each computer block will produce two graphs, a computer schedule and a monitor graph, and the network block will produce

```
function [exectime, data] = eventController(seg, data)
switch (segment),
   case 1,
     ttWait('input_event');
     exectime = 0.0;
   case 2,
     data.y = ttAnalogIn(1);
     data.u = calculateOutput(data.y);
     exectime = 0.002;
   case 3,
     ttAnalogOut(1, data.u);
     updateState(data.y);
     exectime = 0.003;
   case 4,
     ttSetNextSegment(1); % loop
end
```

Figure 13. Example of a code function implementing an event-based controller.

a network schedule. The computer schedule will display the execution trace of each task and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel. In an analogous way, the network schedule shows the transmission of messages over the network. The monitor graph shows which tasks are holding and waiting on the different monitors during the simulation. Generation of these execution traces is optional and can be specified individually for each task, interrupt handler, and monitor.

6.3 The Wired and Wireless Network Blocks

The network blocks are event-driven and execute when messages enter or leave the network. A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

In the wired network block, it is possible to specify the transmission rate, the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA), and a number of other parameters, see Fig. 14. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by a hardware interrupt.

Currently, The TrueTime wireless network block simulates medium access and packet transmission, i.e., the medium access control (MAC) sub-layer of the IEEE 802.11 reference model. 802.11b is used in many laptops and mobile devices of today, and is because of its heavy use a good candidate to include in a simulator. The 802.11b standard is quite complex and is described in [1] and [2]. Our model simulates:

- Direct sequence spread spectrum (DSSS) physical layer (PHY). Other non supported PHY layers of 802.11 are frequency hopping spread spectrum (FHSS) and infra red (IR).

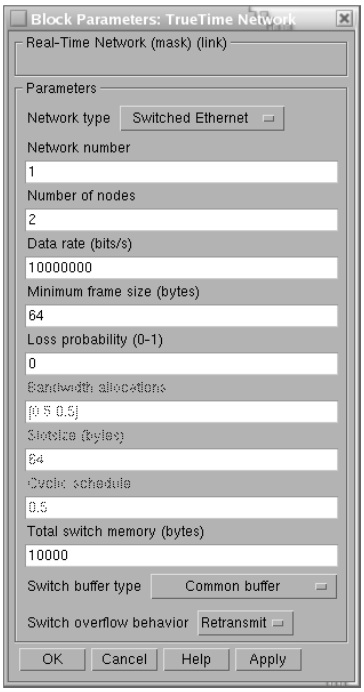


Figure 14. The dialogue of the TrueTime Wired Network block.

- Ad-hoc wireless networks, as opposed to infrastructure-based ones.
- Isotropic antenna.
- Unable to send and receive at the same time.
- Path loss of radio signals modeled as $\frac{1}{d^a}$ where d is the distance in meters and a is a suitably chosen parameter to model the environment.
- The mandatory basic access method based on CSMA/CA.
- Interference from other terminals (shared medium).
- ACK messages on the MAC protocol level.

6.4 Example: Networked Control System

As an example of simulation in TrueTime, we again turn our attention to the networked control system. Using TrueTime, general simulation of the distributed control system is possible wherein the effects of scheduling in the CPUs and simultaneous transmission of messages over the network can be studied in detail. TrueTime allows simulation of different scheduling policies of CPU and network and experimentation with different compensation schemes to cope with delays.

The TrueTime simulation model of the system contains one computer block for each node and a wired network block. The time-driven sensor node contains a periodic task, which at each invocation samples the process

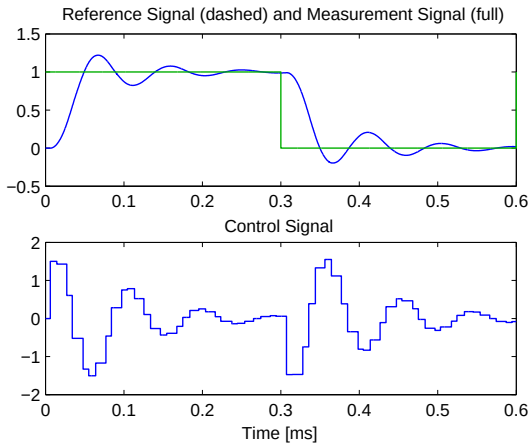


Figure 15. Control performance with interfering network messages and interfering task in the controller node.

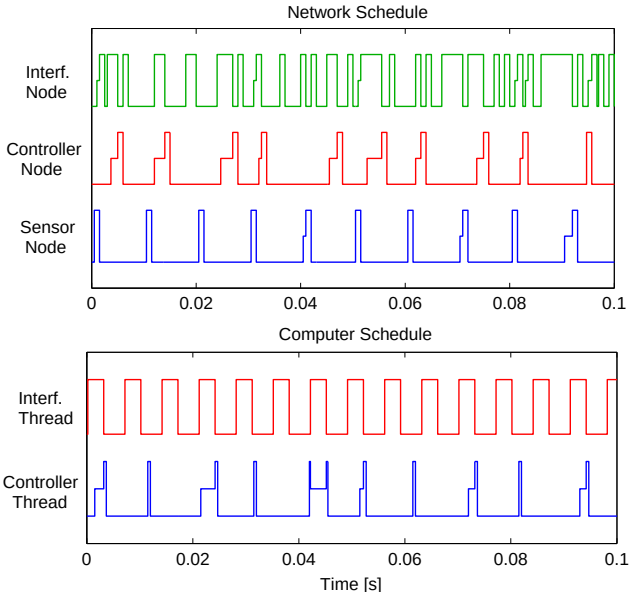


Figure 16. Close-up of schedules showing the allocation of common resources: network (top) and controller node (bottom). A high signal means sending or executing, a medium signal means waiting, and a low signal means idle.

and sends the sample to the controller node over the network. The controller node contains an event-driven task that is triggered each time a sample arrives over the network from the sensor node. Upon receiving a sample, the controller computes a control signal, which is then sent to the event-driven actuator node, where it is actuated. Finally, the interference node contains a periodic task that generates random interfering traffic over the network.

Experiments

In the following simulations, we will assume a CAN-type network where transmission of simultaneous messages is decided based on priorities of the packages. The PD controller executing in the controller node is designed a 10-ms sampling interval. The same sampling interval is used in the sensor node.

We assume that the execution time of the controller is 0.5 ms and the ideal transmission time from one node to another is 1.5 ms. The ideal round-trip delay is thus 3.5 ms. The packages generated by the disturbance node have high priority and occupy 50% of the network bandwidth. We further assume that an interfering, high-priority task with a 7-ms period and a 3-ms execution time is executing in the controller node. Colliding transmissions and preemption in the controller node will thus cause the round-trip delay to be even longer on average and time varying. The resulting degraded control performance can be seen in the simulated step response in Fig. 15. The execution of the tasks in the controller node and the transmission of messages over the network can be studied in detail (see Fig. 16).

Finally, a simple compensation is introduced to cope with the delays. The packages sent from the sensor node are now time-stamped, which makes it possible for the controller to determine the actual delay from sensor to controller. The total delay is estimated by adding the expected value of the delay from controller to actuator. The control signal is then calculated based on linear interpolation among a set of controller parameters precalculated for different delays. Using this compensation, better control performance is obtained, as seen in Fig. 17.

7 Conclusion

This paper has discussed the relationships between control design and the real-time scheduling, and how implementation-level timing variations can be handled in the control design. For embedded applications with limited computing resources, this type of implementation-aware control is especially important. Designing a real-time control system is essentially a codesign problem. Choices made in the real-time design will affect the control design and vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions that must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. Using an analysis tool such as Jitterbug, one can quickly assert how sensitive the control loop is to slow sampling rates, delay, jitter, and other timing problems. Aided by this information, the user can proceed with more detailed, system-wide real-time and control design using a simulation tool such as TrueTime.

References

[1] ANSI/IEEE Std 802.11, 1999.

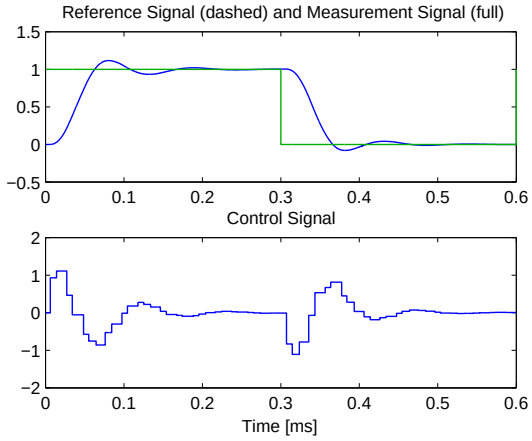


Figure 17. Control performance with delay-compensation.

[2] IEEE Std 802.11b, 1999.

[3] P. Albertos and A. Crespo. Real-time control of non-uniformly sampled systems. *Control Engineering Practice*, 7:445–458, 1999.

[4] Ö. Askerdal, M. Gäfvert, M. Hiller, and N. Suri. Analyzing the impact of data errors in safety-critical control systems. *IEICE Transactions on Information and Systems*, E86-D(12), Dec. 2003. Special Issue on Dependable Computing.

[5] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems*. Prentice Hall, Upper Saddle River, NJ, third edition, 1997.

[6] A. Benveniste and G. Berry. The synchronous approach to real-time programming. *Proceedings of the IEEE*, 79:1270–1282, 1991.

[7] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, 2000.

[8] A. Cervin. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 4–10, York, U.K., June 1999.

[9] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[10] A. Cervin and B. Lincoln. Jitterbug 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7604--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, Jan. 2003.

[11] A. Cervin, B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Göteborg, Sweden, Aug. 2004.

[12] J. El-khoury and M. Törngren. Towards a toolset for architectural design of distributed real-time control systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, England, December 2001.

[13] M. Gäfvert, B. Wittenmark, and Ö. Askerdal. On the effect of transient data-errors in controller implementations. In *Proceedings of the American Control Conference*, pages 3411–3416, Denver, CO, 2003.

[14] T. Hägglund. A predictive PI controller for processes with long dead times. *IEEE Control Systems Magazine*, 12(1):57–60, 1992.

[15] H. Hanselmann. Implementation of digital controllers—A survey. *Automatica*, 23(1):7–32, 1987.

[16] D. Henriksson and A. Cervin. TrueTime 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7605--SE, Department of Automatic Control, Lund Institute of Technology, Oct. 2003.

[17] D. Henriksson, A. Cervin, and K.-E. Årzén. TrueTime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002.

[18] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[19] C.-Y. Kao and B. Lincoln. Simple stability criteria for systems with time-varying delays. *Automatica*, Sept. 2004. To be published.

[20] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Härbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Dordrecht, 1993.

[21] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[22] B. Lincoln. Jitter compensation in digital control systems. In *Proceedings of the 2002 American Control Conference*, May 2002.

[23] B. Lincoln and A. Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, NV, Dec. 2002.

[24] J. Liu and E. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23(1), Feb. 2003.

[25] P. Marti, G. Fohler, K. Ramamritham, and J. M. Fuertes. Jitter compensation for real-time control systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.

[26] J. Nilsson, B. Bernhardsson, and B. Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 34(1):57–64, 1998.

[27] L. Palopoli, G. Lipari, G. Lamastra, and L. Abeni. An object-oriented tool for simulating distributed real-time control systems. *Software—Practice and Experience*, 32:907–932, 2002.

[28] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms*. Kluwer Academic Publishers, Dordrecht, 1998.

Adaptation des Applications Distribuées à la Qualité de Service du Réseau de Communication

Fabien Michaut, Francis Lepage
CRAN CNRS UMR 7039
Faculté des Sciences et Techniques - BP 239
54506 Vandoeuvre Cedex
{fabien.michaut, francis.lepage}@cran.uhp-nancy.fr

Abstract

Le fonctionnement des applications distribuées est tributaire de la Qualité de Service (QoS) offerte par le réseau de communication. Lorsqu'une maîtrise complète de la QoS du réseau est impossible, il peut être pertinent d'adapter en ligne l'application aux ressources disponibles. Ce papier propose un tutorial des techniques d'adaptation des systèmes distribués. Il explique également que la bonne mise en oeuvre de ces techniques implique leur intégration au sein d'une architecture de QoS. Enfin, nous présentons une nouvelle architecture de QoS, l'architecture QoS-Adapt, et un exemple d'application distribuée adaptative.

1 Introduction

Aujourd'hui, les échanges d'information prennent des dimensions géographiques plus importantes. De ce fait, les réseaux utilisés sont le plus souvent hétérogènes et non déterministes. Parallèlement, les informations transférées sont de plus en plus complexes (voix, vidéo, données robotiques, etc.) et contraintes (temps-réel, etc.). Ainsi, les réseaux de communication à commutation de paquets initialement conçus pour acheminer des trafics sans contraintes particulières (messagerie, transfert de fichiers, etc.) doivent à présent satisfaire les besoins d'applications distribuées "critiques".

Parmi ces nouvelles applications, les plus nombreuses sont sans doute les applications multimédias. Celles-ci se distinguent des applications "classiques" par le fait qu'elles manipulent essentiellement des flux continus et non des fichiers[1]. De par leur nature, les flux multimédias sont contraints en terme de QoS[2] (contraintes temporelles fortes entre les différentes unités qui composent un flux par exemple).

L'usage croissant d'Internet par des applications de télé-opération doit aussi être souligné. Ces applications consistent à déporter le pilotage de systèmes automatisés. Pour ces applications, les informations échangées sont des ordres de commande et des mesures issues de capteurs et sont fortement contraintes. Ainsi, l'insertion d'un réseau dans la boucle de commande perturbe le fonctionnement du système, nuit à ses performances et peut menacer sa sécurité et celle de son environnement.

Remarquons que le contrôle de systèmes distribués n'est pas un concept nouveau. Depuis plus de 20 ans, les réseaux locaux industriels permettent de commander et de surveiller de tels systèmes. Toutefois, ces réseaux relèvent totalement de leur exploitant et présentent une faible dispersion géographique. Ce dernier peut donc faire une évaluation préalable des ressources nécessaires puis architecturer et dimensionner son réseau en conséquence. L'exploitant a une maîtrise complète du réseau, ce qui n'est pas le cas lorsque le réseau employé est géré tout ou partie par un opérateur extérieur.

Pour prendre en compte les contraintes de QoS des applications, il existe plusieurs approches. Celles-ci peuvent se classer en deux catégories.

La première repose sur la gestion des ressources, par réservation ou par gestion de priorités. L'avantage majeur de l'approche par réservation de ressource est de fournir une garantie stricte de QoS. Son inconvénient principal réside dans la complexité et la lourdeur des mécanismes qu'elle nécessite. En conséquence, la réservation de ressource ne résiste pas au facteur d'échelle. L'approche par gestion de priorités se contente, quant à elle, d'aménager le "best-effort" et est incapable d'offrir de garantie stricte de QoS.

La seconde approche considère qu'aucune garantie ne peut être obtenue ni du système terminal, ni du réseau de communication. C'est alors à l'application de s'adapter à son environnement [14].

Notons que ces deux approches ne sont pas antagonistes mais complémentaires [5]. C'est le cas par exemple lorsqu'une réservation de ressources est effectuée et qu'un ajustement des demandes de l'application aux ressources disponibles est nécessaire.

[1] Le terme fichier est utilisé ici pour désigner un ensemble de données temporellement indépendantes les unes des autres.

[2] Qualité de Service.

Dans la deuxième partie de ce papier, nous présentons les travaux de la littérature qui s'inscrivent dans la deuxième catégorie d'approches, c'est-à-dire les travaux qui visent à doter les applications distribuées de moyens leur permettant de pallier l'insuffisance du système de communication en adaptant leur exécution. Nous expliquons, dans la troisième partie, que la bonne intégration des techniques d'adaptation dans les systèmes distribués requiert différents mécanismes particuliers que nous explicitons, au sein d'une architecture à QoS intégrée. Enfin, nous présenterons l'architecture QoS-Adapt [29] dans la quatrième partie.

2 Adaptation dans les systèmes distribués

2.1 Adaptation en Automatique: commande adaptative (d'après [26])

La "Commande adaptative" est un ensemble de techniques utilisées pour l'ajustement automatique en ligne et en temps réel des régulateurs des boucles de commande afin de réaliser ou maintenir un certain niveau de performances quand les paramètres du procédé à commander sont soit inconnus soit/et varient dans le temps. Son principe est schématisé sur la figure 1.

Le cadre pour l'adaptation donné par la théorie de la commande adaptative est rarement utilisé pour implanter des techniques d'adaptation dans les systèmes distribués. En effet, le choix et la mesure d'un indice de performance est difficile pour ce type de système . Ensuite, la modélisation des systèmes de bout en bout par des modèles continus est très rarement réaliste. Enfin, **la nature distribuée des systèmes concernés ne permet pas de disposer d'une vue globale instantanée du système**.

Il n'existe donc pas de cadre conceptuel générique pour l'adaptation des systèmes distribués. Par contre, plusieurs techniques spécifiques ont été proposées. Ces techniques d'adaptation se caractérisent par le niveau du système auquel elles sont implantées [16]. Le paragraphe suivant les présente en les classant selon le niveau auquel elles interviennent.

2.2 Adaptation au niveau système (système terminal et système de communication)

CONTROLE DE FLUX

La technique d'adaptation de niveau transport la plus connue est celle mise en oeuvre dans le protocole TCP où la source module son débit en fonction des pertes de paquets. Le rôle de cette adaptation est d'assurer le contrôle de flux. Notons de plus que l'algorithme de retransmission de TCP est aussi adaptatif. Les valeurs de temporisation utilisées sont déterminées en fonction du délai aller-retour de la connexion concernée.

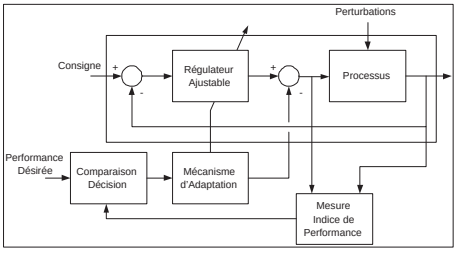


Figure 1. Commande adaptative

CHOIX ET/OU ADAPTATION DU PROTOCOLE

Une technique d'adaptation simple consiste à choisir le protocole le plus adapté à chaque communication. Ce choix peut être statique ou dynamique selon l'horizon temporel de travail désiré. Dans le premier cas, le choix est effectué au début et pour toute la durée de la communication. Des exemples de ce type d'adaptation sont donnés dans [46] et [48] où la pile protocolaire est choisie (statiquement) selon les besoins de l'application. Dans le premier cas, le choix du protocole peut être remis en cause périodiquement s'il est nécessaire de s'adapter sur des horizons plus courts. Le système pourrait ainsi être capable de changer dynamiquement de protocole selon les variations de QoS du réseau.

Par exemple, le débit utile d'une connexion peut être amélioré aux dépens de sa fiabilité. Une solution consiste à basculer du protocole TCP au protocole UDP [17]. Il est aussi possible d'employer des connexions à fiabilité partielle, ou POC (Partial Order Connection), ce qui permet de disposer d'une plus grande flexibilité entre les connexions à fiabilité nulle (par exemple UDP) et celles à fiabilité "totale" (par exemple TCP).

ORDONNANCEMENT DES PAQUETS

Une autre stratégie consiste à adapter les techniques d'ordonnancement des paquets de données aux réseaux utilisés: lors de l'emploi de réseaux haut débit pour la transmission continue de flux à forte contrainte temporelle, l'ordonnancement des paquets est directement lié aux risques de congestion et à la capacité de maintenir une QoS. Il est effectivement impératif de transmettre les données dans l'ordre afin d'éviter les stockages en file d'attente. Ceux-ci sont nécessaires aux traitements de remise en ordre des paquets [16] et pour assurer la continuité du flux. De plus, à haut débit, les buffers se remplissent à grande vitesse. Dans le cas de réseaux RTC, le trafic peut être organisé en utilisant des techniques de *batching* afin de réduire les temps de latence et d'optimiser le coût de la connexion (les données peuvent être ordonnancées pendant les périodes d'inactivité) [41].

TAILLE DES PAQUETS

Un protocole de communication peut aussi s'optimiser par le choix de la taille des unités de données à transférer selon le réseau utilisé [16]. En effet, la taille des paquets influe directement sur les temps de mise en file d'attente et de traitement par les routeurs. Modifier la taille des paquets est une opération très délicate car selon le principe d'ALF[3] [11] l'application doit organiser les données en unités logiques de traitement. Le contenu des paquets impose donc des contraintes relatives à la taille de ceux-ci.

CHOIX DU RESEAU

[3] Application Level Framing

Dans une configuration où plusieurs réseaux sont disponibles (GSM, WAN, RTC), il peut être intéressant de choisir le(s) réseau(x) le(s) plus adapté(s) aux contraintes QoS associées aux données. Les informations à transmettre sont démultiplexées dans le cas où plusieurs réseaux sont utilisés à la fois, ce qui impose une classification des données pour déterminer celles à envoyer sur chaque réseau [16].

Les adaptations de niveau protocole sont compliquées à mettre en place: une relation très étroite entre l'application et les couches réseau est impérative. Le protocole doit prendre en compte la nature des informations pour s'adapter.

ORDONNANCEMENT DANS LES SYSTEMES D'EXPLOITATION

Certaines approches ont proposé des techniques d'ordonnancement des tâches permettant l'adaptation des applications aux ressources disponibles du système terminal (processeur, mémoire, etc.) [6] [12] [22] [39] [40]. Dans [6], les applications sont supposées proposer différents niveaux d'exécution. Un gestionnaire de QoS décide du niveau d'exécution qu'une application doit utiliser en fonction de l'utilisation courante du processeur. Dans [22], une analyse du respect des contraintes temporelles de l'application est réalisée a priori. Les applications en sont notifiées et peuvent alors adapter leur comportement en conséquence. Dans ces travaux, l'adaptation n'est pas réalisée dans le système d'exploitation mais au niveau applicatif. Par contre, le système d'exploitation offre des fonctionnalités propices à l'adaptation de l'application.

2.3 Adaptation au niveau *middleware*

Les techniques dites de niveau *middleware* consistent à implémenter des services entre les applications et le système. Il existe trois approches majeures [16]. La première est le filtrage: un service intercepte les données applicatives avant transmission et les transcode ou augmente leur niveau de compression. Cette technique est utilisée dans [41] et [43] et le concept est déjà proposé dans [15]. Dans [17] une technique similaire est utilisée, mais les compressions et transcodages sont réalisés par des proxys sur le réseau.

La deuxième approche possible est d'implémenter des services qui stockent et placent dans un cache des données pendant les périodes de bonne connectivité. Les données sont ensuite réordonnées au moment de leur utilisation et pendant les périodes de mauvaise connectivité [41].

Enfin, la troisième approche consiste à intégrer au client un proxy local capable d'émuler le serveur dans un mode de fonctionnement dégradé pendant les périodes de mauvaise connectivité [41].

2.4 Adaptation au niveau application

Dans la plupart des cas, ce sont les applications elles-mêmes qui sont les plus aptes à réagir pour faire face aux changements de QoS du réseau. L'adaptation consiste alors à ajuster les besoins des applications aux possibilités offertes par le réseau. Ce type d'adaptation est une solution valable et utile lorsqu'une maîtrise complète de la QoS du système sous-jacent de bout en bout est impossible.

CHANGEMENT DE NATURE DE L'INFORMATION

Une technique d'adaptation simple consiste à changer la nature de l'information. Si la transmission d'une photographie n'est pas possible, il est peut être envisageable de transmettre une description de la photographie sous forme textuelle.

MODULATION DU DEBIT

Les techniques d'adaptation les plus connues reposent sur la modulation du volume d'information émise sur le réseau. Elles sont particulièrement adaptées aux cas des médias continus (vidéo, audio) pour lesquels il est possible de réduire le débit en modifiant le taux de compression des codecs utilisés [7] [13] [44] ou en changeant de codec [17] en fonction de la qualité de réception.

MEMOIRES-TAMPON DE TAILLE VARIABLE

Dans le cas des médias continus et plus particulièrement des applications audio, il est souvent impératif de recevoir les informations de façon régulière. La périodicité de réception des données peut être perturbée par la variation du délai (ou gigue) des paquets. Pour s'affranchir des problèmes de gigue, des mécanismes de compensation de délai sont mis en place au niveau du récepteur: les données sont stockées dans des mémoires-tampons avant d'être interprétées. Ces mémoires-tampon (buffers de play-out) sont dimensionnées de façon à fluidifier le flux d'information et ainsi éliminer les problèmes de gigue. Ce type de mécanismes d'adaptation de mémoires-tampon à la gigue est proposé par exemple dans les applications d'audioconférence vat [21] et Free Phone [47].

CODAGE HIERARCHIQUE

Les applications multimédias peuvent utiliser des techniques de codage hiérarchique pour s'adapter: dans ce cas, l'information est codée en plusieurs couches. Une couche de base contient l'information codée à basse qualité. Les autres couches sont facultatives, et complètent la couche de base en lui ajoutant des détails (*enhancement layer*). Quand le réseau n'est pas chargé, toutes les couches sont transmises, et l'information est reconstruite avec un bon niveau de qualité. Dans le cas contraire, les couches additionnelles sont éliminées du réseau.

TECHNIQUES PLUS GENERALES

D'autres approches plus générales proposent des bibliothèques de composants génériques. Un composant générique est une entité logicielle destinée à un traitement particulier (compresseur vidéo, etc.). Les composants incluent des interfaces dédiées à leur paramétrage. Le développeur construit alors l'application avec ces composants. Le comportement de l'application peut alors être modifié en changeant la configuration de chaque composant (choix des paramètres d'un compresseur vidéo par

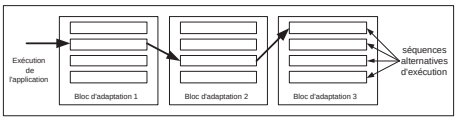


Figure 2. Blocs d'adaptation

exemple) [9] [4] [19] [23] [24] [40].

Le code de l'application peut aussi se décomposer en plusieurs blocs. Un bloc correspond à une tâche particulière de l'application et est associé à un niveau donné de QoS. Il est alors possible de modifier le comportement de l'application en choisissant dynamiquement les blocs à exécuter. L'application propose ainsi plusieurs chemins d'exécution (cf. figure 2) [2] [9] [35]et doit fournir les fonctions de transition permettant de basculer d'un chemin d'exécution à un autre [9].

Les différentes configurations des composants et les chemins d'exécution correspondent à des profils d'exécution de l'application et sont prévus pour être utilisés selon des profils prédéfinis d'utilisation des ressources [9].

STRATEGIES DE DISTRIBUTION DES TACHES

Enfin, dans le cas d'applications distribuées coopératives, la distribution géographique des tâches conditionne l'utilisation des différents chemins employés et des ressources locales des sites distincts [23]. Une technique d'adaptation consiste à modifier la distribution géographique des tâches qui s'exécutent sur les différents systèmes terminaux pour s'adapter à la disponibilité des ressources sur les liens et les systèmes terminaux concernés.

REMARQUE

Dans tous les cas, les mécanismes d'adaptation au niveau applicatif doivent être prévus dès la conception de l'application. Mais il s'agit des techniques qui offrent l'adaptabilité la plus souple [16].

2.5 Adaptation au niveau utilisateur

Le niveau final d'adaptation est l'adaptation du comportement de l'utilisateur. L'utilisateur doit être conscient de l'impact de ses actions sur le fonctionnement du système terminal et du système de communication [16]. Par exemple, lors d'un transfert de fichier, l'affichage d'une barre de progression permet à l'utilisateur de juger s'il souhaite ou non continuer l'opération.

Le système doit être chargé d'informer l'utilisateur des problèmes rencontrés et lui proposer une (des) configuration(s) alternative(s) plus adaptée(s) que celle(s) définie(s) dans ses préférences [15].

3 Le besoin d'une architecture à QoS intégrée

La mise en place des techniques d'adaptation dans le contexte de systèmes distribués implique de nombreuses contraintes. Par exemple, l'objectif étant de s'adapter aux ressources disponibles, il est indispensable de définir des mécanismes de mesure et de surveillance de la disponibilité des ressources. De plus, le comportement de l'application doit convenir aux attentes de l'utilisateur (niveau ultime de QoS). La QoS perçue par celui-ci résulte de la coopération de tous les niveaux du système distribué (application, système d'exploitation et réseau). Il faut donc tenir compte de la QoS à chaque niveau du système, c'est-à-dire des ressources utilisées jusqu'aux préférences de l'utilisateur. Les adaptations des différentes applications qui s'exécutent sur le même terminal doivent être coordonnées, contrairement aux approches les plus courantes, où chaque application réalise individuellement des mesures de QoS et pilote sa propre adaptation. L'absence de coordination peut par exemple poser des problèmes de stabilité des mécanismes d'adaptation ou de compétition dans l'utilisation des ressources partagées (processeur, énergie, etc.).

L'implantation de mécanismes d'adaptation de l'application requiert ainsi une architecture à QoS intégrée. Dans ce type d'architecture, la QoS est prise en compte et spécifiée à tous les niveaux (de l'utilisateur aux couches les plus basses). L'architecture doit éviter à l'utilisateur et au développeur d'applications de se préoccuper de la complexité de la gestion intégrée de la QoS. L'architecture pilote globalement les adaptations des différentes applications.

L'objectif de cette partie est de passer en revue les contraintes qu'une architecture de QoS doit respecter et les mécanismes qu'elle doit inclure.

3.1 La spécification de la QoS

La QoS exprime les niveaux de performance requis par les systèmes distribués. Les performances globales du système dépendent des performances individuelles de chaque niveau du système. Par exemple, les performances du réseau ne déterminent pas à elles seules les performances du système de bout en bout. Dès lors, il est nécessaire de spécifier les besoins et les performances à chaque niveau de l'architecture [38] [40]. Dans la suite, on décrit les paramètres pour les niveaux utilisateur, application, système d'exploitation et système de communication.

PARAMETRES UTILISATEUR

L'utilisateur a besoin de connaître les différentes configurations disponibles de l'application. A ce niveau, la QoS est spécifiée en terme de préférences utilisateur. Ces préférences sont un ensemble de paramètres compréhensibles et interprétables par l'utilisateur: combinaison de paramètres objectifs (taille de l'image) et subjectifs (qualité de l'image). L'utilisateur doit pouvoir aussi préciser ses préférences relatives à l'adaptation. Par exemple, il peut souhaiter privilégier la qualité du son sur celle de l'image dans le cas d'une vidéoconférence si la qualité de la session doit être dégradée. L'utilisateur peut agir sur le système pour l'informer s'il juge le niveau de qualité de la session inacceptable, et si, en conséquence,

une nouvelle adaptation est nécessaire.

La nature de ces paramètres étant souvent subjective, la diversité des utilisateurs, des services et des perceptions très grande, la saisie des paramètres utilisateur n'est pas facile et nécessite une interface appropriée. Des exemples intéressants d'interfaces sont donnés dans [36] et [40].

Les paramètres de QoS utilisateur sont différents selon les applications. Dans tous les deux cas, la spécification de la QoS peut inclure le coût du service[4] (l'utilisateur doit pouvoir fixer des limites [15], [41], [43]).

PARAMETRES APPLICATION

La spécification de la QoS au niveau application est une traduction de la QoS utilisateur en termes quantitatifs. Elle permet d'exprimer les performances attendues de l'application et reçues par celle-ci.

Au niveau application, la spécification de la QoS correspond à une configuration applicative. Elle est réalisée par le développeur de l'application.

La QoS application est très souvent spécifiée par des caractéristiques du média manipulé par l'application (débit du flux, délai de bout en bout, synchronisation inter/intra média). C'est le cas dans [37] par exemple. Cette spécification correspond plutôt à la QoS attendue du service de transport sous-jacent.

PARAMETRES SYSTEME D'EXPLOITATION

A ce niveau, les ressources mises à contribution sont le(s) processeur(s), la mémoire, les bus d'entrées/sorties, et les périphériques multimédias. Comme la ressource la plus limitée est le processeur, la majorité des recherches sur la QoS dans les système d'exploitation a été effectuée sur les techniques d'ordonnancement des tâches sur le processeur [10].

La QoS peut être par exemple spécifiée par un taux d'occupation du processeur [6] ou une "bande passante processeur" [10] qui correspond au pourcentage de la ressource processeur totale. Il est aussi possible d'exprimer la QoS par le temps de traitement sur le processeur alloué périodiquement par un couple (c, t) où c est le temps processeur alloué chaque t seconde. Dans [38], les besoins en ressources processeur d'une tâche et l'allocation des mémoires-tampons sont exprimés par la priorité, le début, la deadline, la durée, la période, etc. L'ordre des tâches est également précisé.

PARAMETRES SYSTEME DE COMMUNICATION

Les paramètres de QoS du système de communication utilisés communément sont la bande passante, le délai, la variation du délai et le taux d'erreurs:

Les applications requièrent la bande passante correspondante au débit du trafic qu'elles génèrent. Une bande passante insuffisante provoque une dégradation de la QoS.

Le délai de bout en bout tient compte des différents délais (codage, paquétisation, propagation, transmission, commutation, temps passé en file d'attente, etc.).

La variation du délai de bout en bout (ou gigue) cause des problèmes de synchronisation entre l'émetteur et le récepteur. La gigue résulte de la mise en file d'attente des paquets dans les nœuds intermédiaires du réseau.

La fiabilité du service de communication est exprimée par le taux de pertes de paquets.

Remarque: ces paramètres et les techniques de mesure associées sont définis en détail dans [34].

Notons que les applications (et leurs trafics associés) ont des caractéristiques et des contraintes de QoS différents. Les paramètres pertinents de QoS sont différents selon le "type" d'application et les mécanismes d'adaptation proposés. Cela est particulièrement évident au niveau applicatif et reste vrai à toutes les couches de l'architecture. Il est donc nécessaire d'adapter la spécification de la QoS à la "nature" de l'application. Dans l'hypothèse où on chercherait à fournir des techniques génériques de spécification de la QoS, une classification des applications et des flux est indispensable. Les rares travaux effectués en ce sens concernent les flux, ou du moins caractérisent les applications en fonction des flux qu'elles manipulent (c'est le cas dans [18] par exemple).

3.2 La translation de la QoS

Comme expliqué dans le paragraphe précédent, il est nécessaire de spécifier les besoins à chaque niveau de l'architecture. Toutefois, les paramètres de QoS des couches basses (inférieures à la couche application) ne sont pas compréhensibles pour la majorité des utilisateurs et des développeurs. Les besoins de l'application sont souvent difficiles à évaluer pour le développeur [1], [15]. Par exemple, les débits de la plupart des applications vidéo sont variables et la spécification d'un taux d'erreurs maximum acceptable est souvent impossible (niveau de redondance de l'information fonction du codec utilisé et du taux de compression potentiellement variable).

Dans ce contexte, il est plus simple d'obtenir suffisamment d'informations de la part de l'utilisateur et/ou de l'application (exprimées respectivement en termes utilisateur et applicatif) et de convertir cette spécification en paramètres de QoS pour les couches sous-jacentes. L'opération de conversion de la spécification d'un niveau à un autre est appelée translation ou *mapping* [8] [37].

Dans les cas les plus simples, cette conversion peut se limiter à copier directement des paramètres à une certaine couche. Elle peut aussi consister à utiliser des procédures spécifiques (convertir le rafraîchissement, la taille de l'image et la résolution en valeurs de bande passante requise au niveau transport par exemple).

Le service de translation doit être bidirectionnel [38]: la conversion des paramètres du haut vers le bas de l'architecture permet de déterminer la QoS requise par l'utilisateur à tous les niveaux de l'architecture. Il est aussi nécessaire d'informer l'utilisateur de la QoS disponible (en terme utilisateur) en connaissant la QoS disponible

[4] La spécification doit inclure le prix que l'utilisateur est prêt à payer pour le service sinon le niveau maximum de celui-ci est toujours demandé [8].

aux niveaux bas de l'architecture. La propriété bidirectionnelle de la translation est difficile à assurer car cette dernière est souvent ambiguë [9] [37]. Par exemple, il est aisé de calculer la bande passante réseau nécessaire au transfert d'une séquence vidéo caractérisée par son rafraîchissement et la taille des images. Lorsque le calcul est difficile (compressions à débit variable par exemple), il est possible d'utiliser des tables de conversion obtenues empiriquement [40]. La translation inverse est plus délicate car, à une valeur de bande passante, peut correspondre plusieurs couples de valeurs (taille des images, rafraîchissement). Une solution à ce problème consiste à utiliser des règles de conversion supplémentaires prédéfinies [37].

Etant donné la diversité des applications et des environnements d'exécution (systèmes d'exploitation et réseaux), il n'existe pas d'algorithme générique de translation. L'approche proposée dans [9] consiste à mesurer les performances de l'application dans toutes les conditions possibles de disponibilité des ressources[5]. De cette façon, une base de données des performances par rapport aux ressources disponibles est construite. Elle permet par la suite d'en déduire les performances de l'application en se basant sur la mesure des ressources disponibles courantes et en interpolant les mesures stockées dans la base.

3.3 La surveillance (métrologie) de la QoS

L'objectif de l'architecture étant de permettre aux applications d'adapter leur exécution à la QoS disponible, des mécanismes de surveillance des ressources sont indispensables. Cette partie présente les niveaux de l'architecture auxquels ces mécanismes peuvent intervenir. Les techniques de surveillance utilisées dans les schémas adaptatifs proposés dans la littérature sont ensuite énumérés.

NIVEAUX DE SURVEILLANCE

Les mesures peuvent être obtenues à tous les niveaux de l'architecture. Plus l'information est obtenue à un niveau élevée, moins elle est précise, et moins une coopération étroite entre les couches est nécessaire [5].

La surveillance des ressources au niveau application fournit une vision boîte noire du système de communication et du système d'exploitation. Pour cela, les systèmes terminaux peuvent par exemple s'échanger des informations régulièrement (par l'intermédiaire de RTP/RTCP par exemple). Il est impossible de faire la différence des dégradations des performances dues aux ressources réseau de celles dues aux ressources système en utilisant cette approche.

Au niveau transport, les algorithmes de contrôle de congestion peuvent être mis à profit pour avoir des informations sur la bande passante puisqu'ils cherchent à adapter le débit le plus justement possible en évitant de sous-utiliser la bande passante et de provoquer des congestions. Ces algorithmes utilisent des informations implicites telles que les pertes de paquets, le délai, etc.

De la même façon, les ressources disponibles sur le système terminal peuvent conditionner le fonctionnement des applications. Dans [6] et [22], la mesure des ressources processeur disponibles est utilisée pour notifier à l'application que les ressources requises ne sont pas disponibles et que cette dernière doit s'adapter en conséquence.

TECHNIQUES DE SURVEILLANCE

Les protocoles Real Time Protocol (RTP) et Real Time Control Protocol (RTCP) [42] sont utilisés dans de nombreux mécanismes d'adaptation pour s'informer de la QoS du réseau. C'est le cas des mécanismes d'adaptation du débit [7] [13] [44] et d'adaptation des mémoires-tampon de lecture (vat [21] et Free Phone [47]). Une session RTP combine deux flux d'informations: un flux de données (un flux audio par exemple) et un flux de paquets de contrôle. Les paquets de contrôle constituent le flux RTCP. Chaque participant à une session RTP diffuse périodiquement aux autres participants un rapport RTCP. Ces rapports indiquent directement (ou permettent de déterminer selon le paramètre) le taux de perte, la gigue, le délai aller-retour.

Une autre approche consiste à doter les objets et librairies utilisés pour programmer les applications d'interfaces dédiées à la surveillance de la QoS. C'est le cas dans [5] où l'API des sockets a été étendue par une fonction get_bw() qui retourne une valeur de bande passante calculée en se basant sur la valeur de la fenêtre de congestion de TCP. Dans [9] et [23], les objets ou composants utilisés pour le développement des applications incluent des métriques qui mesurent les performances de l'objet ou du composant. Ces métriques sont accessibles via des interfaces dédiées.

Enfin, des mesures peuvent être réalisées par observation et analyse du trafic utile de l'application (mesure passive). Cette approche nécessite d'avoir une connaissance précise des caractéristiques du trafic de l'application. Lorsque ce n'est pas le cas, il est possible de réaliser des mesures en envoyant des "paquets-sonde" sur le réseau (mesure active). Ce flot de mesure circule sur le réseau entre les deux systèmes terminaux concernés et est supposé apprécier la Qualité de Service de bout en bout telle qu'elle est ressentie par l'application. A son arrivée, le flot est analysé et il est alors possible d'en déduire des métriques (délai, pertes de paquet, etc.). Un tutorial des techniques de mesures actives est disponible dans [34].

POLITIQUE DE SURVEILLANCE

Quelles que soient les techniques de surveillance mises en oeuvre, il est nécessaire de définir la période avec laquelle les mesures sont réalisées. Ce paramètre est directement lié à la période du processus d'adaptation. De plus, selon les mécanismes d'adaptation et les applications, il peut s'avérer utile de définir des mécanismes d'alarme (pour notifier qu'un paramètre a dépassé un seuil important pour l'application).

[5] En pratique, les performances sont mesurées pour différents scénarios caractéristiques de disponibilité des ressources en utilisant un émulateur d'environnements d'exécution.

3.4 Le pilotage de l'adaptation

PRINCIPE DE TRANSPARENCE

La complexité de la gestion de la QoS doit être cachée à l'application: les mécanismes précédents de surveillance et de translation de la QoS, ainsi que le pilotage de l'adaptation de l'application doivent lui être externes et délégués à l'environnement d'exécution (architecture).

En effet, la limitation de l'implémentation de ces fonctionnalités dans l'application a des avantages majeurs: tout d'abord, le travail du développeur s'en trouve facilité [24] [4] [9] [2] [25]. Celui-ci n'ayant pas à se soucier des mécanismes parfois complexes de la gestion de la QoS, il n'a par conséquent pas besoin d'avoir des connaissances poussées dans ce domaine. L'environnement de développement doit permettre au développeur de doter l'application de mécanismes pour déclarer ses différentes configurations possibles et les niveaux de QoS associés [8] [9] [25] en terme applicatif. La translation de la QoS est effectuée par l'architecture (cf. 3.2), le développeur spécifie la QoS uniquement en termes applicatifs sans tenir compte des implications de cette QoS aux niveaux inférieurs de l'architecture [8] [24].

L'application se limite à fournir des fonctions de transition qui permettent à l'architecture de faire basculer l'application d'une configuration à une autre [25] sans nécessairement préciser la politique d'adaptation. La politique d'adaptation est l'ensemble des règles qui définissent comment l'application doit s'adapter en fonction de la variation de la QoS.

Finalement, l'externalisation du contrôle de l'application permet à un contrôleur (ou superviseur) de QoS d'avoir une vision globale dans le cas où plusieurs applications s'exécutent sur le même système terminal et d'adapter de façon cohérente l'ensemble des applications.

PRINCIPE DE SEPARATION

La transparence implique la séparation des mécanismes de l'architecture chargées de la gestion de la QoS et du traitement des médias [8]. On retrouve ainsi la séparation des entités et des mécanismes logiciels en deux espaces distincts: un espace de contrôle et un espace applicatif. L'espace de contrôle contient toutes les entités dédiées au contrôle de la QoS. Ces entités sont par exemple des agents de surveillance, gestionnaires de QoS, gestionnaires de ressources, contrôleurs d'admission, etc. L'espace applicatif contient des entités souvent appelées composant ou objet "de flux" ou "de média" qui sont consacrées au traitement de l'information utile de l'application (flux vidéo, audio, etc.).

L'interaction entre les deux espaces (ou plans) est réalisée au travers d'interfaces spécifiques des objets utilisés pour programmer l'application. Dans [40], les objets ne font pas partie des deux espaces à la fois. Par contre, à chaque entité de l'espace applicatif correspond une entité dans l'espace de contrôle (cf. figure 3).

En pratique, les entités de l'espace de contrôle sont implémentées par des librairies spécifiques liées au code de l'application [25]. Les entités et les interfaces de contrôle sont cachées au développeur. De cette façon, les développeurs d'applications n'utilisent que les objets et/ou les interfaces de l'espace applicatif alors que les concepteurs des librairies d'objets et de l'architecture doivent prévoir les deux types d'interface et/ou d'objets [24].

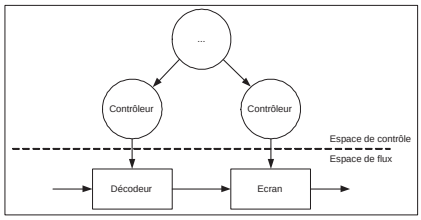


Figure 3. Les objets de l'espace de contrôle et de l'espace de flux inter-opèrent via des interfaces dédiées

3.5 Stabilité de l'adaptation

Comme tout processus dynamique, le mécanisme d'adaptation peut être confronté à des problèmes de stabilité [6]. Le cas d'une application audio est présenté dans [3]. Pour cette application, le choix du codage audio est conditionné par la bande passante disponible. Pour une bande passante inférieure à $48kb/s$, le codage "A" est utilisé. Lorsque la bande passante est supérieure à $48kb/s$, le codage "B" est choisi. Des problèmes d'instabilité peuvent survenir quand la valeur de la bande passante oscille autour du seuil utilisé pour le choix. Ce problème d'instabilité est illustré sur la figure 4 (R: bande passante disponible (b/s), flow rate: débit du flux généré par l'application, $Adm6\_p\_lo$: seuil de décision entre les deux codages). La bande passante disponible (courbe R) oscille autour de $48kb/s$, l'algorithme ne cesse de commuter entre les deux codages (phénomène représenté par le débit en sortie de l'application, courbe en gras "flow rate") et l'oscillation est auto-entretenue.

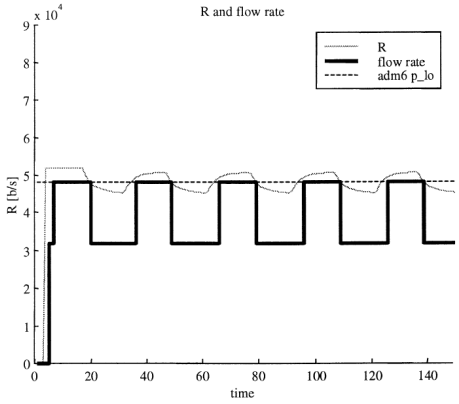


Figure 4. Instabilité dans le choix du codage audio [3]

D'autres problèmes peuvent survenir quand plusieurs applications adaptatives utilisent des ressources communes. C'est le cas des mécanismes d'adaptation du débit. Dans [15], il est montré qu'une concurrence entre les connexions adaptatives se produit et que cela conduit les débits utilisés par chaque application à osciller (une compétition entre ces connexions et des connexions TCP peut aussi se produire [45]). Quand les applications adaptatives sont exécutées sur la même station, le pilotage des adaptations par un superviseur peut remédier à ce type de problèmes [28].

3.6 Mécanismes de contrôle

L'objectif des mécanismes de contrôle de la QoS est de s'assurer que les flux générés par les applications respectent les spécifications annoncées, le plus souvent en terme de débit. Ils sont surtout indispensables lorsque des réservations de ressources sont faites (ce qui n'est pas l'objet de ce papier) car ces dernières sont garanties sous réserve de la conformité du flux à des spécifications précises de débit moyen et de débit crête (on parle alors de "traffic shaping").

Ces mécanismes ont aussi un intérêt dans les schémas adaptatifs. Dans le cas des mécanismes d'adaptation de débit par exemple, ils peuvent servir de boucle de retour pour contrôler le volume des flux générés en boucle fermée.

Dans ce partie, nous avons constaté qu'il n'existe pas de théorie générale pour l'adaptation des applications distribuées. Cependant, de nombreuses techniques d'adaptation propres ont été proposées dans la littérature, et nous les avons passées en revue. Par ailleurs, nous avons expliqué que la bonne intégration de ces techniques dans les systèmes distribués requiert différents mécanismes spécifiques que nous avons détaillés, au sein d'une architecture à QoS intégrée.

4 Une Architecture de QoS pour l'adaptation des applications

L'architecture de QoS proposée [30] [32] adopte une structure en trois couches et intègre un mécanisme d'adaptation similaire au modèle Prayer, basé sur les concepts de classe de QoS, de bloc d'adaptation et d'action d'adaptation.

L'architecture est illustrée sur la figure 5. Les trois couches qui la composent sont: la couche "ressources", la couche "gestion", et la couche "application". Avant de décrire ces différents éléments, nous commençons par présenter le mécanisme d'adaptation utilisé pour des raisons de simplicité. L'architecture proposée sera ensuite décrite: Nous commençons par détailler sa structure statique, puis nous expliquons son fonctionnement dynamique et le processus d'adaptation en ligne de l'application.

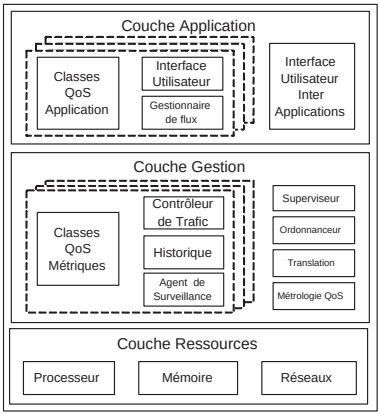


Figure 5. Représentation informelle de l'architecture de QoS proposée

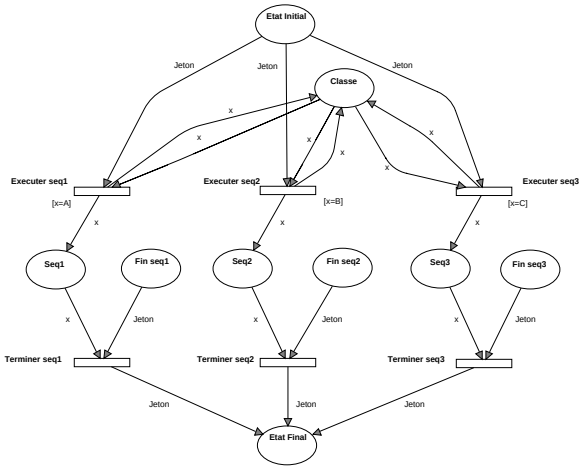


Figure 6. Choix des séquences d'exécution

4.1 Mécanisme d'adaptation

L'interaction entre l'application et la couche gestion des ressources est fondée sur les concepts de classe de QoS et de séquences d'exécution inspirés des travaux de [2]: D'une part, l'application définit des classes de QoS qui sont des niveaux de disponibilité des ressources (ou niveaux de QoS disponible). D'autre part, l'application renferme plusieurs séquences alternatives d'exécution. Une séquence est une portion de code exécutable. Chaque séquence est associée à une classe de QoS. L'application exécute alors la séquence d'exécution correspondante à la QoS disponible.

Le réseau de Petri de la figure 6 illustre l'exemple d'une application pour laquelle trois séquences d'adaptation "1", "2" et "3" sont respectivement associées à trois classes de QoS "A", "B" et "C". Les classes de QoS sont modélisées par trois couleurs de jeton: les couleurs A, B et C. A l'exécution de chaque séquence correspond une place (respectivement les places "Seq1", "Seq2" et "Seq3").

4.1.1 Adaptation en cours d'exécution

Durant l'exécution de la séquence, la couche de gestion se charge de déterminer la classe de QoS la plus adaptée et l'indique à l'application. Si la classe la plus adaptée change, la couche de gestion en notifie l'application laquelle réagit alors au changement en fonction d'actions d'adaptation prédéfinies et associées à la séquence d'exécution. Ces actions sont spécifiées par l'application avant le démarrage du bloc d'adaptation dans une séquence CAP (Classe, Action, Procédure) qui associe une classe de QoS, un couple d'actions d'adaptation et une séquence d'exécution (désignée par le champ procédure).

Le couple d'actions d'adaptation comporte une action à entreprendre lorsque la classe de QoS n'est pas respectée, appelée "ActionBas" pour laquelle il existe plusieurs primitives: BLOCK (l'application suspend son exécution jusqu'à ce que la classe de QoS soit à nouveau disponible), BEST_EFFORT (l'application ignore les changements de QoS et poursuit l'exécution de la séquence en cours), ROLLBACK (l'application recommence l'exécution du bloc d'adaptation courant avec une nouvelle classe de QoS), ABORT (l'application arrête son exécution et quitte le bloc d'adaptation en cours) et SWITCH (permet le passage d'une séquence à une autre en cours d'exécution).

La deuxième action, appelée "ActionHaut", définit l'action à mener lorsqu'une classe de QoS supérieure peut être satisfaite. Il existe plusieurs primitives pour ce type d'actions (les définitions de ces actions sont identiques aux précédentes): BEST_EFFORT, ROLLBACK, ABORT et SWITCH.

4.1.2 Bloc d'adaptation

Une application peut effectuer séquentiellement plusieurs tâches différentes. Pour chaque tâche et pour une même classe de QoS, il peut être intéressant de choisir des actions d'adaptation différentes. Ce cas de figure est pris en considération au travers du concept de bloc d'adaptation. Un bloc d'adaptation est un ensemble de séquences d'exécution (cf. figure 2). Ainsi, pour chaque tâche de l'application, un bloc d'adaptation est défini et à chacune de ses séquences correspond des actions d'adaptation différentes.

Cette méthode d'adaptation permet aux applications de fournir elles-mêmes les règles avec lesquelles elles doivent s'adapter tout en déléguant à l'architecture le soin de piloter l'adaptation. La complexité du processus d'adaptation liée à la gestion des ressources est totalement extérieure aux applications. Le principe de séparation de [8] est respecté (cf. chapitre 2).

4.2 Description de l'architecture

La **couche "ressources"** contient les différentes ressources, c'est à dire les ressources réseau et les ressources propres au système terminal (processeur, mémoire, etc.).

La **couche "gestion"** renferme tous les mécanismes nécessaires à la gestion de la QoS dans l'architecture. Une partie des éléments qui la composent sont uniques et sont utilisés pour la gestion de toutes les applications qui s'exécutent sur un système terminal. Il s'agit du superviseur, de l'ordonnanceur, du composant de translation et du service de métrologie de QoS. Les autres éléments sont associés à une application. Il s'agit des classes QoS métriques, du contrôleur de trafic, de l'historique et de l'agent de surveillance.

Les différents éléments de la **couche "application"** sont l'interface utilisateur inter-applications, les "classes de QoS application", l'interface utilisateur et le gestionnaire de flux. L'interface utilisateur inter-applications est unique et est utilisée pour toutes les applications. Les trois autres éléments sont associés à chaque application qui s'exécute sur le système terminal.

Le superviseur pilote l'adaptation de toutes les applications qui s'exécutent sur le système. Il est chargé de déterminer et d'indiquer à l'application la séquence à exécuter. Il est aussi responsable de l'activation des fonctions d'adaptation durant l'exécution d'une séquence. Le superviseur se base sur les informations délivrées par l'ordonnanceur et le composant de translation d'une part, et par l'agent de surveillance de QoS et l'historique de chaque application d'autre part.

L'ordonnanceur définit les priorités entre les applications en fonction des préférences inter-applications fixées par l'utilisateur. Les préférences de l'utilisateur sont saisies par l'intermédiaire de l'interface utilisateur inter-applications de la couche application.

La QoS est spécifiée en termes applicatifs par le développeur sous la forme de "classes de QoS application". Elles représentent les différentes configurations qu'offre l'application. Ces configurations sont proposées à l'utilisateur par l'intermédiaire de l'interface utilisateur.

Le composant de translation traduit les "classes de QoS application" en termes quantitatifs (métriques) sous la forme de "classes de QoS métriques". Ces dernières sont utilisées par le superviseur pour décider de la politique de surveillance nécessaire et pour déterminer la classe de QoS dans laquelle l'application doit s'exécuter.

Un agent de surveillance est associé à chaque application. Il est chargé de mettre en œuvre la politique de surveillance élaborée par le superviseur. Il s'appuie sur le service de métrologie de QoS. Il lui indique les paramètres à mesurer et la fréquence avec laquelle ils doivent être mesurés. Le service de métrologie est conçu de façon modulaire: les techniques de mesures sont implémentées sous la forme de boîtes appelées "capteurs". N'importe quelle technique de mesure peut y être intégrée a priori. Utilisé entre deux systèmes terminaux, le service de métrologie de l'architecture permet de mesurer plusieurs paramètres différents simultanément et dans les deux sens de la communication. Le service de métrologie n'est pas

décrit ici pour des raisons de place. Pour plus de détails, le lecteur pourra se reporter à [31] et [33].

Un contrôleur de trafic vérifie que le trafic généré par l'application est conforme aux spécifications annoncées. Si ce n'est pas le cas, le gestionnaire de flux (couche application) en est informé et modifie les paramètres de l'application en conséquence. Par exemple, cela peut consister à affiner les réglages d'un codec vidéo qui génère un flux vidéo plus volumineux que prévu.

Un historique renseigne le superviseur sur les classes de QoS dans lesquelles l'application s'est exécutée aux instants précédents.

4.2.1 Fonctionnement

CONFIGURATION DE L'ARCHITECTURE ET DE L'APPLICATION

Lorsqu'une application démarre, elle s'enregistre auprès de l'interface utilisateur inter-applications. L'application crée ensuite son interface utilisateur propre.

Une fois l'interface utilisateur de l'application créée, l'utilisateur est invité à saisir ses préférences. Il commence par indiquer ses préférences inter-applications via l'interface utilisateur inter-application. Les préférences inter-applications sont exprimées sous forme d'un ordre de préférence entre les différentes applications qui s'exécutent sur le système terminal. L'utilisateur saisit ensuite ses préférences pour l'application en utilisant l'interface utilisateur propre à l'application. L'interface lui permet d'exprimer ses préférences concernant les classes de QoS de l'application en les classant par un ordre de préférence. L'utilisateur lance ensuite l'exécution de l'application.

Par ailleurs, l'application s'enregistre auprès du superviseur et exporte ses classes de QoS. A ce stade, les classes de QoS spécifient la QoS à un niveau applicatif. Ces spécifications sont ensuite traduites en spécifications quantitatives sous forme de métriques par le composant de translation. Ce dernier crée alors une "classe de QoS métrique" pour chaque classe de QoS applicative. Une fois la translation réalisée, le superviseur connaît les paramètres de QoS pertinents et instancie un agent de surveillance qui est responsable de l'observation des ressources pour l'application. L'agent démarre ensuite la surveillance à la demande du superviseur.

ADAPTATION EN LIGNE

Avant le démarrage de l'exécution d'un bloc d'adaptation, l'application spécifie un ensemble de séquences CAP. Le superviseur détermine alors la classe de QoS la plus adaptée et démarre la séquence d'exécution correspondante.

Cette classe de QoS est choisie en fonction des ressources disponibles, des préférences de l'utilisateur et de la priorité de l'application. Les ressources disponibles sont déterminées par un agent de surveillance. Les préférences de l'utilisateur sont fournies par l'interface utilisateur. La priorité de l'application est indiquée par l'ordonnanceur. Cette information est déduite des préférences inter-applications. De plus, le superviseur tient compte de la classe de QoS en cours et de l'historique afin d'éviter l'instabilité de l'application (changement permanent de classe de QoS, déclenchement trop fréquent des actions d'adaptation). L'historique indique les classes de QoS dans lequel le système se trouvait précédemment.

Durant l'exécution du bloc d'adaptation, le superviseur continue à déterminer périodiquement la classe de QoS la plus adaptée. Si celle-ci est différente de la classe en cours, l'action d'adaptation correspondante est initiée: selon que la nouvelle classe est supérieure ou inférieure à la classe en cours, le superviseur déclenche l'action "ActionHaut" ou l'action "ActionBas" (cf. figure 7).

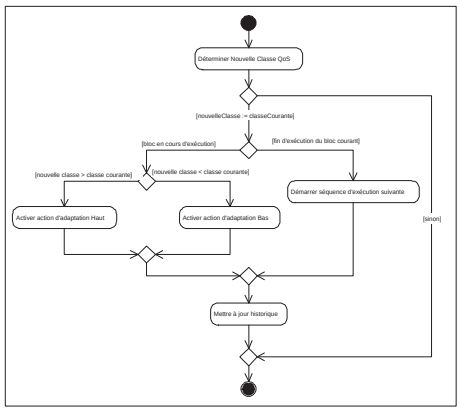


Figure 7. Adaptation en ligne

La période avec laquelle le superviseur détermine la classe de QoS la plus adaptée (ou période d'adaption) dépend de l'application: selon l'application, l'exécution est découpée en blocs d'adaptation périodiques ou non-périodiques.

5 Application à la télé-opération d'un robot mobile

L'architecture présentée précédemment est mise en œuvre sur une plateforme de télé-opération d'un robot mobile. Cette plateforme (cf. figure 8) est composée d'un robot mobile, d'un ordinateur et d'une caméra embarqués et d'un poste de commande. Le robot et son ordinateur embarqué sont interconnectés par un port série. L'ordinateur embarqué est connecté au réseau par une technologie sans fil 802.11b.

L'objectif est d'asservir la position x du robot mobile depuis le poste de commande (cf. figure 9). Le robot est commandé en vitesse. Il mesure sa position x grâce à ses capteurs (odomètres) et la retourne au poste de commande. Une commande en vitesse v est calculée sur le poste de commande puis elle est transmise au robot. Le correcteur est de type proportionnel k.

L'utilisation du réseau pour les transmissions de la commande et de la mesure (position du robot) introduit un retard dans la boucle d'asservissement.

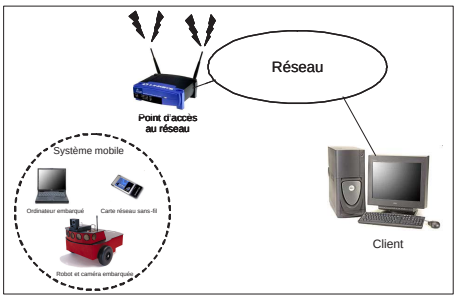


Figure 8. Plateforme d'expérimentation

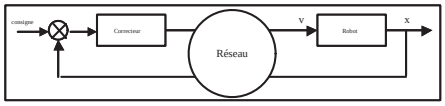


Figure 9. Boucle d'asservissement

5.1 Impact du retard sur l'asservissement

Dans la suite, le délai du réseau est considéré symétrique (délai identique à l'aller et au retour). Le délai aller-retour, c'est à dire le retard global de la boucle, est noté rtt (round-trip time).

Cette étude est réalisée en simulation avec Matlab. La fonction de transfert du robot a été préalablement identifiée à l'aide de la boîte d'identification CONTSID présentée dans [20]. On obtient la fonction suivante pour la réponse en vitesse:

$$F(s) = \frac{16,6117.e^{-70.10^{-3}s}}{s^3 + 4,0450s^2 + 16,5611s} \qquad (1)$$

La réponse indicielle du système est simulée pour différents retards qui provoquent des dépassements (cf. figure 10).

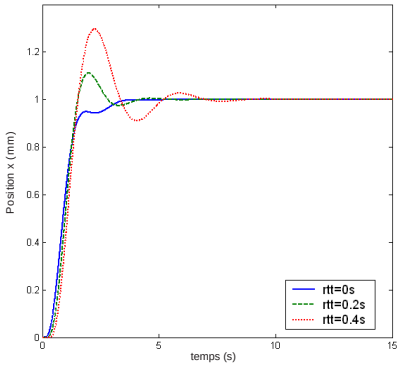


Figure 10. Réponse indicielle pour plusieurs valeurs de retard

Le dépassement est très gênant dans les cas où le robot se déplace à proximité d'obstacles (mur par exemple), et peut provoquer des collisions.

5.2 Stratégie pour supprimer le dépassement

La solution la plus simple pour supprimer le dépassement consiste à calculer la valeur du gain de la boucle d'asservissement pour le délai maximum.

Cette approche présente l'inconvénient majeur de ralentir le système: Par exemple, si l'on désire supprimer les dépassements en présence de retards inférieurs ou égaux à $1s$ (soit en choisissant un gain $k = 0,3$ par exemple), le temps de montée passe de $1,38s$ (avec un gain de 0.9) à $6,53s$ (cf. tableau 1).

k	$0,3$	$0,5$	$0,7$	$0,9$
t_m	$6,53$	$3,59$	$2,37$	$1,38$

Table 1. Temps de montée en fonction du gain (valeurs obtenues pour rtt=0)

Se contenter de fixer une valeur de gain pour un retard maximum n'est donc pas satisfaisant. En effet, pour les situations dans lesquelles le retard est très inférieur à la valeur maximum, le système est inutilement lent.

Il serait alors intéressant de choisir la valeur du gain selon des plages définies de retards.

Le tableau 2 présente les retards maxima calculés pour un dépassement nul, pour différentes valeurs de k.

k	$0,3$	$0,5$	$0,7$	$0,9$
$rtt_{\max}$	$1,10$	$0,50$	$0,30$	$0,10$

Table 2. Retards maxima (dépassement nul) en fonction de k

Remarque: Ces valeurs sont très élevées et pour certaines très supérieures aux délais couramment observés aujourd'hui sur l'Internet. Ceci est dû à l'extrême lenteur du système étudié. Une étude similaire d'un système plus rapide (robot se déplaçant à plusieurs m/s par exemple) conduirait à obtenir des retards maxima beaucoup plus faibles et à l'échelle des délais observés sur l'Internet.

5.3 Mise en œuvre de l'architecture

L'application est divisée en deux parties: une partie est située sur le poste de commande, elle est chargée de recevoir la mesure de la position du robot, de calculer la commande et de l'envoyer au PC embarqué. L'autre partie, localisée sur le PC embarqué, a pour rôle de récupérer la position du robot, d'envoyer cette information au poste de commande, de recevoir la commande du poste de commande et de la faire suivre au robot. L'application ne dialogue pas directement avec les capteurs et les actionneurs du robot mais avec sa carte-contrôleur qui est un système de commande fermé dont nous n'avons pas la maîtrise (cf. figure 11).

L'application est développée selon le modèle de l'architecture QoS-Adapt. Pour cela, quatre classes de

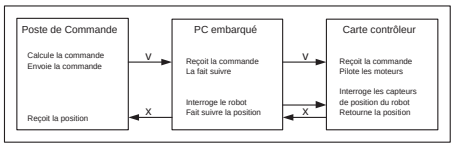


Figure 11. Structure de l'application

QoS (numérotées de 1 à 4) sont définies pour la partie de l'application située sur le poste de commande. Chaque classe correspond à une configuration applicative, c'est à dire à une valeur de gain k (respectivement 0,9, 0,7, 0,5 et 0,3), elle est destinée à être utilisée selon la valeur mesurée du retard.

On définit un bloc d'adaptation. Ce bloc est répété périodiquement. Il est constitué de quatre séquences d'exécution. Chaque séquence consiste à calculer la commande durant la période du bloc, pour chaque classe de QoS. Ici, à une classe de QoS ne correspond qu'une séquence d'exécution. La période d'un bloc d'adaptation est de $2s$.

Trois actions d'adaptation différentes sont implémentées: BEST_EFFORT (l'application ignore les changements de QoS, la commande continue à être calculée avec le même gain), SWITCH (l'application commute d'une séquence à une autre en cours d'exécution. La valeur de k est modifiée en cours d'exécution du bloc), BLOCK (l'application suspend son exécution jusqu'à ce que la classe de QoS soit à nouveau disponible. Le robot est immobilisé).

Durant l'exécution d'un bloc, la classe de QoS la plus adaptée est recalculée à une période de $0,5s$.

Les séquences CAP sont présentées dans le tableau 3.

Classe de QoS	ActionHaut	ActionBas
1	-	SWITCH
2	BEST_EFFORT	SWITCH
3	BEST_EFFORT	SWITCH
4	BEST_EFFORT	BLOCK

Table 3. Séquences CAP

Remarques: Aucune action "Haut" n'est définie pour la séquence correspondant à la classe de QoS 1 car aucune classe de QoS n'est supérieure à celle-ci. De plus, l'action BEST_EFFORT est aussi utilisée pour les actions "Haut" afin d'éviter les oscillations entre deux séquences différentes durant l'exécution d'un bloc lorsque le rtt varie autour d'une limite d'une classe de QoS. Quand le délai aller-retour est supérieur à $1,1s$, aucune des classes de QoS spécifiées n'est disponible. Dans ce cas, l'application bascule dans la séquence associée à la classe la plus basse (la classe 4) si elle n'y était pas et l'action d'adaptation Bas est activée: Ici, le système se bloque. Si le délai aller-retour est à nouveau inférieur à $1,1s$ avant la fin du bloc, l'application reprend l'exécution de la séquence associée à la classe de QoS 4.

Le bloc, les séquences et les actions d'adaptation sont modélisé par le réseau de Petri temporisé de la figure 12.

Les classes de QoS sont modélisées par quatre couleurs de jeton: les couleurs A, B, C et D associées respectivement aux classes 1, 2, 3 et 4. Pour prendre en compte les cas dans lesquels aucune des classes de QoS n'est disponible, nous définissons la couleur supplémentaire E. L'exécution d'une séquence est représentée par la présence d'un jeton de couleur correspondante dans la place "Seq": Par exemple, l'exécution de la séquence associée à la classe de QoS 1 est modélisée par un jeton de couleur A dans la place "Seq".

La couleur du jeton présent dans la place "Classe" indique la classe de QoS courante. La couleur du jeton présent dans la place "Nouvelle Classe" donne la classe de QoS la plus adaptée et conditionne l'activation où non des actions d'adaptation. Contrairement aux modèles présentés précédemment, la présence d'un jeton dans cette place n'exprime pas nécessairement que la classe de QoS la plus adaptée est différente de la classe de QoS courante (il est possible d'avoir simultanément un jeton de couleur x dans la place "Classe" et dans la place "Nouvelle Classe").

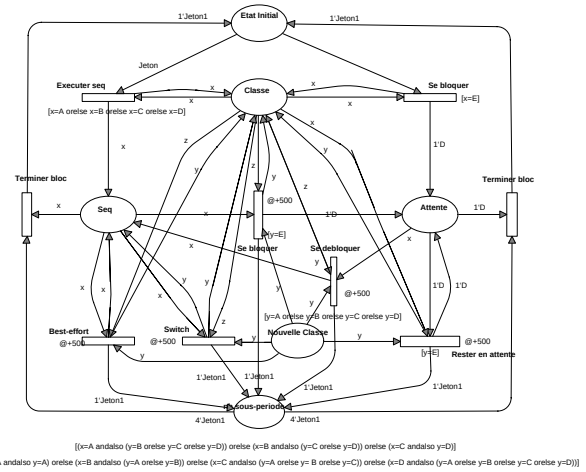


Figure 12. Réseau de Petri du bloc d'adaptation

Au démarrage de l'application, un jeton est présent dans la place "Etat initial" et la classe de QoS est indiquée par la présence d'un jeton de couleur correspondante (A, B, C, D ou E) dans la place "Classe". Selon la couleur de ce dernier, une seule des transitions "Exécuter seq" et "Se bloquer" est franchissable (cf. figure 12). Suite au tir de celle-ci, un jeton est mis dans la place correspondante ("Seq" ou "Attente").

La période d'un bloc d'adaptation est $2s$. Durant l'exécution d'un bloc, la classe de QoS la plus adaptée est recalculée par la couche de gestion qui l'indique en introduisant un jeton de couleur correspondante dans la place "Nouvelle Classe" à une période de $0,5s$. Autrement dit, la couleur du jeton présent dans la place "Seq" est remise

à jour périodiquement (toutes les $0,5s$) lorsqu'une action d'adaptation SWITCH ou BEST_EFFORT doit être initiée. Lorsque l'action BLOCK doit être enclenchée, le jeton de la place "Seq" est supprimé, et un jeton D est positionné dans la place "Attente". Le jeton ne peut quitter cette place que lorsque la nouvelle classe de QoS est A, B, C ou D ou si l'exécution du bloc est terminée. La fin de l'exécution du bloc (fin de la période de $2s$) est indiquée par la présence de quatre jetons dans la place "nb sous-période", c'est-à-dire après quatre calculs de la classe de QoS la plus adaptée.

Remarques: Les durées de sensibilisation sont indiquées en ms. Les deux gardes indiquées en bas de la figure sont associées respectivement aux transitions Switch et Best-effort.

5.4 Résultats expérimentaux

Les résultats présentés sont ceux obtenus avec le système réel (cf. figure 8). Les retards sont générés par un émulateur de réseau [27].

La figure 13 présente les résultats obtenus pour une expérience au cours de laquelle le rtt varie.

La consigne et la réponse du système en fonction du temps sont représentées sur la figure du haut. La figure du milieu indique la valeur du rtt mesurée durant l'expérience. La valeur du gain en fonction du temps est représentée sur la figure du bas.

On vérifie bien que la valeur du gain k change en fonction du rtt mesuré,

Le système suit la consigne et semble correctement identifié,

Les résultats expérimentaux montrent l'intérêt de notre approche: l'adaptation de la loi de commande à la QoS du réseau permet d'éviter les dépassements du système. Les risques de collision du robot avec son environnement sont alors limités. La stratégie utilisée d'adaptation de la loi de commande n'a pas la prétention d'être la meilleure. Elle a été choisie pour sa simplicité. Les possibilités offertes par l'architecture (notamment une métrologie plus complète que celle utilisée ici) pourraient permettre la mise en œuvre de stratégies plus élaborées et ainsi l'amélioration des performances de l'asservissement.

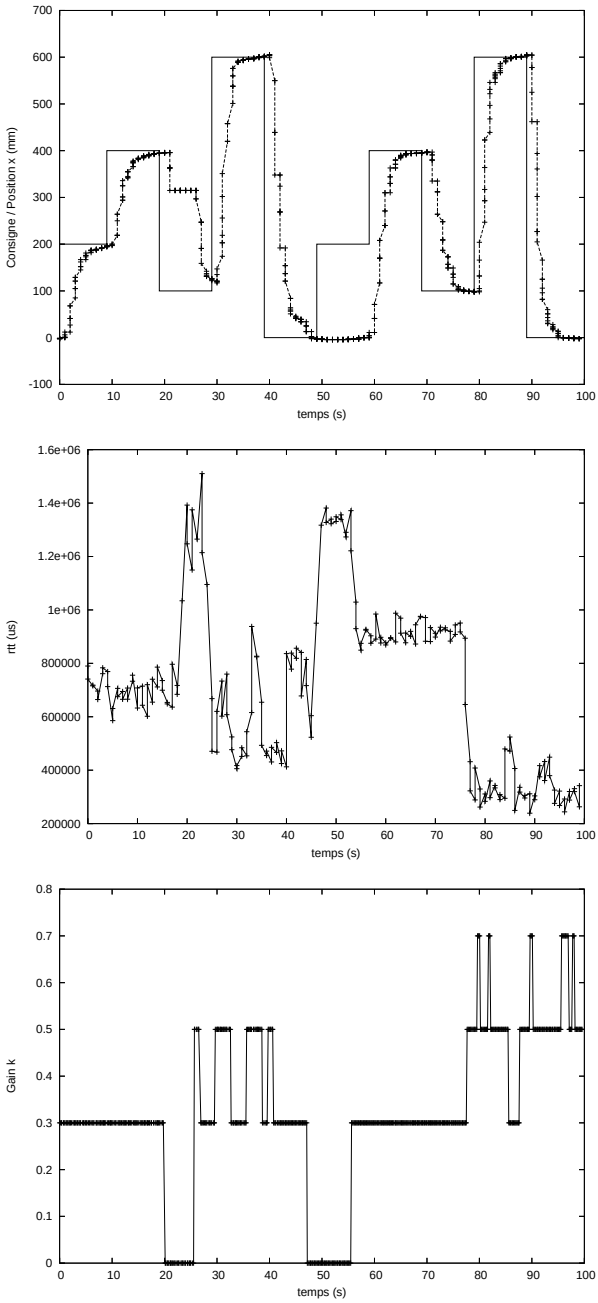


Figure 13. Résultats expérimentaux

6 Conclusion et perspectives

Durant ces dix dernières années, les applications mettant à contribution des réseaux de communication à commutation de paquets se sont complexifiées: les réseaux "grande distance" initialement conçus pour acheminer des trafics sans contraintes particulières (messagerie, transfert de fichiers, etc.) doivent aujourd'hui satisfaire les besoins d'applications critiques dont les trafics sont fortement contraints en terme de délai, de bande passante, etc. Le fonctionnement de ces applications est donc directement conditionné par la QoS fournie par le réseau de communication, qu'il est indispensable de prendre

en considération. Pour cela, deux approches concourantes sont envisageables: La première repose sur la gestion des ressources, par réservation ou par gestion de priorités. La réservation dynamique ne résiste pas au facteur d'échelle et la gestion des priorités n'offre pas de garantie. L'essentiel des travaux proposés dans la littérature s'inscrivent dans cette approche. Partant de l'hypothèse qu'une maîtrise du système de communication n'est pas totale ou est impossible, la seconde permet de considérer que l'application peut, dans une certaine mesure, s'adapter à la variation de la QoS si elle connaît cette variation.

Les travaux présentés dans ce papier se positionnent dans la seconde approche. Nous avons présenté un tutorial des techniques d'adaptation dans les systèmes distribuées. Nous avons montré que la bonne mise en oeuvre de ces techniques implique de prévoir un certain nombre de mécanismes de QoS. Ces mécanismes peuvent être rassemblés au sein d'une architecture à QoS intégrée.

Dans ce contexte, nous avons introduit une nouvelle architecture, l'architecture QoS-Adapt, qui permet l'adaptation en ligne des applications. L'application à la télé-opération d'un robot mobile a permis de vérifier le bon fonctionnement de l'architecture. Les expérimentations ont montré que l'architecture rend possible la mise en œuvre d'une stratégie pour s'adapter aux retards introduits par le réseau dans la boucle de commande et ainsi éviter par exemple les dépassements du système.

Dans la perspective de ces travaux, nous pensons qu'il serait intéressant d'établir une classification des applications. Ceci permettrait d'associer à chaque classe un modèle de spécification de la QoS et des fonctions de translation. Cela contribuerait aussi à définir de façon plus précise une politique adaptée de surveillance de la QoS: paramètres pertinents à mesurer, période de mesure.

References

[1] Y. Bao. *Quality of Service control for real-time multimedia applications in packet-switched networks*. PhD thesis, University of Delaware, May 1998.

[2] V. Bharghavan and V. Gupta. A framework for application adaptation in mobile computing environments. In *Proc. IEEE Compsac'97*, Nov. 1997.

[3] S. N. Bhatti and G. Knight. Enabling QoS adaptation decisions for Internet applications. *Computer Networks*, 31:669–692, 1999.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. Saikoski. A principled approach to supporting adaptation in distributed mobile environments. In *Proc. 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, Limerick, Irlande, June 2000.

[5] J. Bollinger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 24(5):376–390, May 1998.

[6] S. Brandt, G. Nutt, T. Berk, and M. Humphrey. Soft real-time applications execution with dynamic quality of service assurance. In *Proc. International Workshop on Quality of Service*, Napa, USA, 1998.

[7] I. Busse, B. Deffner, and H. Schulzrinne. Dynamic QoS Control of Multimedia Applications based on RTP. In *Proc. First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russie, June 1995.

[8] A. T. Campbell. *A Quality of Service Architecture*. PhD thesis, Computing Department, Lancaster University, 1996.

[9] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. Technical Report TR1999-793, Oct. 1999.

[10] F. Cheong and R. Lai. QoS Specification and mapping for distributed multimedia systems : a survey of issues. *The Journal of Systems and Software*, 45:127–139, 1999.

[11] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. ACM SIGCOMM*, 1990.

[12] I. Demeure. Une contribution à la conception et à la mise en ouvre d'applications sous contraintes de QoS temporelles, réparties, adaptables. Université des Sciences et Technologies de Lille, Dec. 2002.

[13] C. Diot. Adaptative Applications and QoS Guaranties. In *Proc. IEEE Multimedia Networking*, Aizu, Japon, Sept. 1995.

[14] C. Diot, C. Huitema, and T. Turletti. Multimedia applications should be adaptive. In *Proc. HPCS'95, Mystic (CN)*, 1995.

[15] C. Diot and A. Seneviratne. Quality of service in heterogeneous distributed systems. In *Proc. 30th Hawaï International Conference on System Sciences*, Hawaï, USA, Jan. 1997.

[16] A. Friday, N. Davies, G. Blair, and K. Cheverst. Developing adaptative applications : the MOST experience. *Integrated Computer-Aided Engineering*, 6(2):143–157, 1999.

[17] M. Fry and A. Ghosh. Application level active networking. *Computer networks*, 31:655–667, 1999.

[18] R. Gopalakrishnan and G. Parulkar. Efficient quality of service support in multimedia computer operating systems. Technical Report WUCS-94-26, Washington University, Nov. 1994.

[19] A. Hafid and G. von Bochmann. Quality-of-Service adaptation in distributed multimedia applications. *Multimedia Systems*, 6(5):299–315, 1998.

[20] E. Huselstein, H. Garnier, A. Richard, and P. Young. La boÓte ‡ outils CONTSID d'identification de modËles ‡ temps continu - Extensions rÈcentes. In *Proc. ConfÈrence Internationale Francophone d'Automatique*, Nantes, France, July 2002.

[21] V. Jacobson. vat x11 based audio conferencing tool, unix manual page, Feb. 1993.

[22] M. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints : Efficient, predictable Scheduling of Independent Activities. In *Proc. 16th ACM Syposium on operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[23] P. Keleher, J. K. Hollingsworth, and D. Perkovic. Exposing application alternatives. In *Proc. International Conference on Distributed Computing Systems*, pages 384–392, 1999.

[24] S. Khan, K. Li, and E. Manning. Padma: An architecture for adaptive multimedia systems. In *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 1997.

[25] K. Lakshman and R. Yavatkar. Integrated CPU and Network I/O QoS Management in an End-System. *Computer Communications Journal, Special Issue on Quality of Service in Distributed Systems*, 21, Apr. 1997.

[26] I. Landau and L. Dugard. *Commande adaptative, aspects pratiques et théoriques*. Editions Masson, 1986.

[27] V. Lecuire, T. Maurey, and F. Lepage. Un Èmulateur de l'Internet rÈalisÈ sous RTLinux. In *Proc. 9th RTS'2001*, Paris, France, Mar. 2001.

[28] B. Li, D. Wu, K. Nahrstedt, and J. Liu. End-to-End QoS Support for Adaptative Applications Over the Internet. In *Proc. Internet Routing and QoS*, Boston, USA, 1998.

[29] F. Michaut. *Adaptation des applications distribuées à la Qualité de Service fournie par le réseau de communication*. PhD thesis, Université de Nancy I, Nov. 2003.

[30] F. Michaut and F. Lepage. A QoS Architecture for Application Execution Adaptation. In *Proc. SNPD'02 ACIS 3rd International Conference on Software Engineering Artificial Intelligence, Networking and Parallel/Distributed Computing*, Madrid, Espagne, June 2002.

[31] F. Michaut and F. Lepage. A Tool to Monitor the Network Quality of Service. In *Proc. NET-CON'2002, IFIP-IEEE Conference on Network Control and Engineering*, Paris, France, Oct. 2002.

[32] F. Michaut and F. Lepage. An Architecture for Application Adaptation based on QoS Monitoring. In *Proc. SMC'02, IEEE International Conference on Systems, Man and Cybernetics*, Hammamet, Tunisie, Oct. 2002.

[33] F. Michaut and F. Lepage. Networks Quality of Service Monitoring. In *Proc. IAR Annual Meeting*, Grenoble, France, Nov. 2002.

[34] F. Michaut and F. Lepage. Application-oriented network metrology : Metrics and active measurement tools. *IEEE Communications Surveys & Tutorials*, 2005.

[35] P. Moghé and A. Kalavade. Terminal QoS of Adaptive Applications. *Bell Labs Technical Journal*, 1998.

[36] K. Nahrstedt. *An architecture for End-to-End Quality of Service Provision and its experimental validation*. PhD thesis, University of Pennsylvania, 1995.

[37] K. Nahrstedt and J. M. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[38] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE Computer*, 29(5):52–63, May 1995.

[39] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. 16th Symp. Operating Systems Principles*, pages 184–197, Oct. 1997.

[40] M. Ott, G. Michelitsch, D. Reininger, and G. Welling. An architecture for adaptive QoS and its application to multimedia systems design. *Computer Communications*, 21(4):334–349, Apr. 1998.

[41] A. Schill, S. Kümmel, T. Springer, and T. Ziegert. Two approaches for an adaptive multimedia transfer service for mobile environments. *Computer & Graphics*, 23:849–856, 1999.

[42] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, Request For Comments RFC 1889, Jan. 1996.

[43] R. D. Silva, B. Landfeldt, S. Ardon, A. Seneviratne, and C. Diot. Managing application level quality of service throught tomten. *Computer Networks*, 31:727–739, 1999.

[44] D. Sisalem. End-to-end quality of service control using adaptative applications. In *Proc. IFIP 5th International Workshop on QoS (IWQoS'97)*, New York, USA, May 1997.

[45] D. Sisalem. Fairness of adaptative multimedia applications. In *Proc. International Conference on Communications (ICC'98)*, Atlanta, USA, June 1998.

[46] A. Soudani. *Integration des fonctionnalités multimédias dans les systèmes répartis temps réels : Application au système de communication industrielle*. PhD thesis, Université du Centre, Monastir et Université Henri Poincaré, Nancy, Feb. 2003.

[47] A. Vega-Garcia. *Mécanisme de contrôle pour la transmission de l'audio sur l'Internet*. PhD thesis, Université de Nice-Sophia Antipolis, INRIA, Oct. 1996.

[48] Q. Wu. *Contribution à l'intégration de communications multimédias dans un environnement MMS*. PhD thesis, Université Henri Poincaré, Nancy, June 1997.

Some Real-Time Issues in Wireless Sensor Networks

David Simplot-Ryl
IRCICA/LIFL, Univ. Lille 1
CNRS UMR 8022, INRIA Futurs
Cité Scientifique, Bât. M3
59655 Villeneuve d'Ascq Cedex, FRANCE
David.Simplot@lifl.fr

Abstract

Wireless sensor networks (WSN) consist of a set of wireless sensor nodes which sense the environment and route the sensed information to a sink node. In this paper, we give an overview of real-time problems in such networks. In particular, we will consider the layers of the communication stack with a focus on MAC and network layer and on two WSN specific tasks which are data gathering and activity scheduling.

1. Introduction

Recent developments in micro-electro-mechanical systems (MEMS), digital electronics and wireless communications have enabled wireless sensor networks which are deployed to collect useful information from an area of interest [1, 2]. Wireless sensor networks (WSN) are expected to be used in a wide range of applications, such as military surveillance, environmental monitoring, target tracking, etc.

A sensor network is a set of nodes in which a battery, a sensing and a wireless communication device are embedded. Sensor networks are a special case of ad hoc networks with objects generally densely deployed either very close or inside a studied phenomenon. Sensor nodes are deployed over hostile or remote environments to monitor a target area. Therefore, their irreplaceable batteries mean that energy is the most important system resource. Moreover, these objects are expected to work and collaborate as long as possible in order to send their collected data to one or more sink stations. These sinks, also called monitoring stations, are considered to have non-limited battery and aim to collect information from sensor nodes in multi-hop manner.

Because WSN applications such as home or site surveillance or patient monitoring, consist in dealings with a physical environment, data communication in sensor networks has timing constraints in term of delays and of end-to-end deadlines. In network area, we talk of QoS (Quality of Service) which can include time constrainst, bandwith, security, robustness, etc.

In the remaining of the paper, we give an overview of last recent techniques proposed to manage communication delays. In Section 2, we consider Medium Access Control solutions which aim to distinguish different packet priorities. In Section 3, we describe multi-hop considerations which correspond to classical network layer. In Section 4, we are interested in real-time aspects of data gathering and activity scheduling.

2. Medium Access Control Layer

Of course, each protocol which aims to enhance performances of MAC layer in wireless ad hoc networks also contributes to reduce packet delivery delays. However, it is not the goal of this paper to give a survey of MAC protocols and the reader can find an up-to-date survey in [3]. Concerning real-time MAC layer, we distinguish two families: non-deterministic and deterministic protocols.

2.1. Non-Deterministic Protocols

The first class contains random-access protocols like Aloha and its variations including IEEE 802.11 (aka WiFi) [4] and IEEE 802.15.4 (aka ZigBee) [5]. These protocols are based on random wait period before the transmission of the next packet. Most of variations of these protocols consist in the introduction of several packet queues which correspond to different packet priorities. A packet of a given queue is treaten if and only if all queues of highest priority are empty. This trick simply deals with 'internal' differentiation and in order to solve 'external' differentiation it suffices to give different maximum wait period for each queue (highest priority with smallest wait period) and on multiplication factor used in case of collision. An illustration of this kind of protocols can be found in [6].

2.2. Deterministic Protocols

The second family contains contention-free protocols like TDMA-based protocols where each sensor owns a given time slot where it is the lonely node which can transmit during this slot. The difficulty of such algorithm is the initialization during which collision can occur. For instance, S-MAC protocol uses this principle [7]. For unique queue layers, it is easy to evaluate time needed to

access to the medium since it is bounded by to the length of the frame multiplied by number of packets in the queue. The problem of such protocols is that initialization phase can be time and energy consuming. Moreover, they fit well with static networks and are not efficient when nodes are moving or even die.

2.3. Priority vs Deadlines

For both families, packet priorities can be indicated with a deadline. In this case, we can use a simple EDF (Earliest Deadline First) for the packet scheduling [8]. This solution can be combined with class-based scheduling [9]. The main problem is the computation of the priority which may take into account multi-hop aspects of the packet delivery instead of point-to-point delays. For instance, the RAP protocol [10] proposes to determine packet speed based on geographical information. This example shows that it is difficult to consider real-time aspect without using cross-layer aspects, in particular between link layer and network layer described in next section.

3. Network Layer

The goal of network layer is to route packets in multi-hop manner in the network. For years numerous routing protocols have been developed for ad hoc networks: Destination-Sequenced Distance-Vector Routing protocol (DSDV) [11], Ad Hoc On-Demand Distance Vector (AODV) [12], Dynamic Source Routing (DSR) [13], Temporally Ordered Routing Algorithm (TORA) [14], Associativity-Based Routing (ABR) [20], and Optimized Link-State Routing Protocol (OLSR) [16]. More protocols and details can be found in the survey of E. Belding-Royer [17]. In general, the existing routing protocols are categorized into two categories: table-driven or on-demand. Table-driven protocols maintain routing information from each node to every other node in the network. Since network may be dynamic, these protocols broadcast updates through the network in order to maintain consistent global network knowledge. On the other hand, on-demand protocol creates routes only when wanted. In this case, flooding is used to broadcast route request. In both cases, flooding or optimized broadcast [18] is used: to disseminate topology information in the case of table driven protocols and to discover routes in the case of on-demand protocols. Recently, position-based routing protocols [19] have taken great attention but they require the setup of a location service in order to find location/position of the destination nodes.

3.1. Table-Driven Routing Protocols

For so-called ‘proactive’ protocols, the consideration of real-time can be treated in disseminated information about the topology network. For instance, in OLSR [16], nodes send only partial information about their neighborhood. Each node selects a subset of its neighborhood which is a set of one-hop neighbors such that each two-hop neighbor can be reach directly by one node of this set. These nodes are called MPR nodes (Multipoint Relays) [20]. It is easy to show that the knowledge of MPR link is enough to compute shortest paths. That is why MPR are used both to minimize the number of topology control messages broadcasted in the network and to reduce the size of these topology control messages.

For our purpose, shortest paths may be inadequate since delay can be different for each depending on current load and environment. Hence, more information has to be sent in topology control messages in order to make to computation of minimum delay path available. In [21], the authors propose a variation of the greedy algorithm used for MPR. Two problems remain in this solution: (1) this algorithm favors the selection of links with small delays but it cannot ensure that path with minimal delay still exist in the disseminated graph; and (2) the variation of MPR is still used for flooding while it is not necessary since MPR is more efficient for flooding than the proposition. In fact, OLSR gives the possibility to declare two distinct sets: the MPR set which is used for flooding – it is one of the more efficient broadcast optimized protocol [18] – and ANS (Advertised Neighbor Set) which contains links we want to declare.

These problems are solved in [22] where a new algorithm is proposed for ANS set. The basic idea is that ANS does not have to ‘cover’ two-hop neighbors like MPR because we do not want to preserve shortest path. Hence, a variation of RNG (Relative Neighborhood Graph) [23] graph reduction combined with the distance measure described in [24] is proposed. The preservation of route with minimum delay is proved and the size of ANS is shown to be smaller than MPR one.

3.2. On-Demand Routing Protocols

Here, the problem is to discover the path with minimum delay when needed. Route discovery is generally performed with piggybacking using flooding. Note that there exist some optimization using caches but it does not impact our discussion. The common solution is to include delay information in route request packet like in ABR [20]. Hence the destination receives several route requests which contain different path and the destination node can select the path with smallest delay. Unfortunately, most of flooding or optimized broadcasts generate messages which are propagated like an increasing circle centered on the source node. The consequence is that only few routes are discovered by such broadcast protocols. In [25], a broadcast protocol designed for multiple-path discovery is proposed. It consists in two phases: (1) optimized flooding to advertise distance to source and (2) flooding with several random sequence numbers from destination to source.

The main difficulty is to find a trade-off between number of messages used for the route request broadcasting and the number of generated routes.

Another way to proceed is to use one optimized broadcast protocol and to favor nodes with small delay. For instance, it is easy to include QoS consideration in the Neighbor Elimination Scheme (NES) [26]. With NES, a relay node does not retransmit the message immediately but waits a given time windows. If during this period all neighbors of the message have been covered by other communication, the node cancels the retransmission of the message. It is straightforward to make the wait period variable of the delay, *e.g.* linear with the delay. Hence, nodes with short delays will retransmit quickly and nodes with long delays will retransmit latter if neighbors are not already covered. With such a solution, we perverse energy efficient broadcasting but we cannot ensure that delay optimal paths are discovered.

3.3. Position-Based Routing Protocols

The task of finding paths in sensor and mobile networks is nontrivial since host mobility and changes in node activity status may cause frequent unpredictable topological changes. If a 'distance' between nodes can be computed, *e.g.* based on geographical position, routing is performed by a scheme that is based on this information. For instance, the MFR (most forward within radius) routing algorithm [28] is a loop-free greedy algorithm: the packet is sent to the neighbor with the greatest progress – the progress of a node is defined as the projection onto the line connecting the current node and the final destination.

In literature, delay is approximated with hop count, and hence solutions attempt to minimize hop count [29]. Note that the hop count can be estimated by using Euclidean distance [30]. Nevertheless, the most promising way of investigations seems to be 'cost-aware' routing algorithms [31, 32]. Let us consider a node U which wants to forward a packet for a node D, the destination. Let $\delta(V)$ be the delay induced by the selection of a neighbor V of U. The portion of progress made with the selection of V is $(UD\text{-}VD)$. If the progress is uniform, there would be $UD/(UD\text{-}VD)$ similar steps, and the total cost would be $\delta(V)\times UD/(UD\text{-}VD)$. As a result, the neighbor V that minimizes $\delta(V)/(UD\text{-}VD)$ will be selected for forwarding the message.

4. Other Issues

There exist some other issues that have been poorly studied because of their novelty. We give here a quick overview of these issues that concern data collection and activity scheduling.

4.1. Data Gathering

Data gathering is an important functionality in WSN [33, 34]. It consists in building a reverse spanning tree which allows to each sensor node to send report to the sink. When several nodes send their report to an identical parent, reports are aggregated in order to minimize number of messages. Most of the current works have been focused on energy efficiency and network lifespan consideration. These techniques generally aim to reduce the number of transmitted packets in order to reduce energy consumption. For our purpose, the problem is to build a tree which minimizes the maximum path delay. This height of the tree gives the period which can be used for reports.

Best centralized solution is the classical MST (Minimum Spanning Tree) algorithm [35] however in order to increase robustness of the structure, there exist some localized variation. For instance, RNG [23] and LMST (Local Minimum Spanning Tree) [36] can be the start structure of the spanning tree. In fact, neither RNG nor LMST are a tree but we can use directed diffusion [37] to transform the graph into a spanning tree. Note that RNG and LMST cannot be used directly with delay since they both require symmetrical weight function.

4.2. Activity Scheduling

As already mentioned, energy is the most important resource in WSN. In order to increase their lifespan, and so the duration of the network, sensors are allowed to turn into sleep mode as soon as they are not required for the monitoring task. Indeed, if too many nodes are deployed over a given surface, only some of them are actually needed for monitoring. The ensuing issue consists in these nodes deciding themselves whether to turn off or not so that the whole area remains fully covered and the subset of active nodes connected.

This is achieved by using 'Connected Area Dominating Sets (CADS) [38] which are connected subsets of nodes which covers the target area. The difficulty is to minimize the size of the CADS. Dominant nodes are periodically changer in order to increase lifespan of the WSN. More details about activity scheduling can be found in [39].

The problem of finding CADS scheduling which delay efficient data gathering or routing is an open problem.

5. Conclusion

We have presented some real-time related works in wireless sensor networks with a focus on link and network layer. To be efficient, most these solutions require time synchronization [40] and delay approximation [41]. These two challenges are of great interest and deserve to be addressed.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey." *Computer Networks* 38 (4), 393–422, 2002.

[2] I. Stojmenovic (ed.), *Handbook of Sensor Networks: Algorithms and Architectures*, Wiley & Sons, 2005.

[3] W. Ye, and J. Heidemann, "Medium Access Control in Wireless Sensor Networks." Chapter in *Wireless Sensor Networks*, T. Znati, K. M. Sivalingam and C. Raghavendra (eds.), Kluwer Academic Publishers, 2004.

[4] G. Anastasi, et al., "IEEE 802.11 in Ad Hoc Networks: Protocols, Performance and Open Issues." Chapter in *Mobile Ad Hoc Networking*, S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic (eds.), Wiley & Sons, 2004.

[5] 802.15.4-2003 IEEE Standard for Information Technology-Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Low Rate Wireless Personal Area Networks (LR-WPANS), 2003.

[6] J.P. Sheu, C.H. Liu, S.L. Wu, and Y.C. Tseng. "A priority MAC protocol to support realtime traffic in ad-hoc networks." *ACM Wireless Networks* 10, 61-69, 2004.

[7] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks." In *Proc. INFOCOM 2002* (New-York, NY, USA, 2002), pp. 1567-1576.

[8] M. Caccamo, L. Y. Zhang, L. Sha, and G. Buttazzo. "An implicit prioritized access protocol for wireless sensor networks." In *Proc. IEEE Real-Time System Symposium (RTSS'02)*, (2002).

[9] A. Mercier, P. Minet, and L. Georges, "A class-based EDF scheduling for Bluetooth piconet", *Ad-Hoc and Sensor Networking: an International Journal*, 2005.

[10] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. "RAP: A real-time communication architecture for large-scale wireless sensor networks." In *Proc. IEEE Real-Time Technology and Application Symposium (RTAS'02)*, (2002).

[11] C. E. Perkins, and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers." In *Proc. ACM SIGCOMM '94*, (2004), pp. 234-244.

[12] C. E. Perkins, and E. M. Royer, "Ad-hoc on-demand distance vector routing." In *Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications*, (1999), pp. 99-100.

[13] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad-Hoc Wireless Networks." Chapter in *Mobile Computing*, T. Imielinske and H. Korth, (eds.), Kluwer, 1996.

[14] V. D. Park and M. S. Corson, "A highly adaptive distributed routing protocol for mobile ad hoc networks," In *Proc. INFOCOM'97*, (1997), pp. 1405-1413.

[15] C. K. Toh, "Associativity based routing for ad-hoc mobile networks." *Wireless Personal Communication* 4 (2), 103-139, 1997.

[16] T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, A. Qayyum, and L. Viennot, "Optimized link state routing protocol." In *Proc. IEEE INMIC* (Pakistan, 2001).

[17] E. Belding-Royer, "Routing Approaches in Mobile Ad Hoc Networks." Chapter in *Mobile Ad Hoc Networking*, S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic (eds.), Wiley & Sons, 2004.

[18] F. Ingelrest, D. Simplot-Ryl, and I. Stojmenovic. "Energy Efficient Broadcasting in Wireless Mobile Networks." Chapter in *Resource Management in Wireless Networking*, M. Cardei, I. Cardei, and D.-Z. Du, Eds, Kluwer. 2005.

[19] S. Giordano, I. Stojmenovic, "Position Based Routing Algorithms for Ad Hoc Networks: A Taxonomy." Chapter in *Ad Hoc Wireless Networking*, X. Cheng, X. Huang and D. Z. Du (eds.), Kluwer, 2004.

[20] A. Laouiti, A. Qayyum, and L. Viennot, "Multipoint relaying: an efficient technique for flooding in mobile wireless networks." In *Proc. 35th Annual Hawaii International Conference on System Sciences (HICSS-34).* (2002).

[21] H. Badis, A. Munaretto, K. Al Agha, and G. pujolle, "QoS for ad hoc networking based on multiple-metric: Bandwidth and delay." In *Proc. IFIP International Workshop On Mobile and Wireless Communications Networks (MWCN'03)*, (Singapore, 2003).

[22] L. Moraru and D. Simplot-Ryl. "QoS preserving topology advertising reduction for OLSR routing protocol for mobile ad hoc networks." submitted.

[23] G. Toussaint, "The relative neighborhood graph of finite planar set," *Pattern Recognition* 12 (4), 261–268, 1980.

[24] J. Cartigny, F. Ingelrest, and D. Simplot. "RNG relay subset flooding protocols in mobile ad-hoc networks." *International Journal of Foundations of Computer Science* 14 (2); 253-265, 2003.

[25] M. Hauspie, J. Carle, and D. Simplot. "Partition detection in mobile ad-hoc networks using multiple disjoint path set." In *Proc. 1st International Workshop on Objects Models and Multimedia technologies (OMMT).* (Geneva, Switzerland, 2003).

[26] W. Peng, X. Lu, "On the reduction of broadcast redundancy in mobile ad hoc networks." In *Proc. Annual Workshop on Mobile and Ad Hoc Networking and Computing (MobiHoc'2000)*, (Boston, MA, USA, 2000), pp. 129–130.

[27] I. Stojmenovic, M. Seddigh, J. Zunic, "Dominating sets and neighbor elimination based broadcasting algorithms in wireless networks." *IEEE Transactions on Parallel and Distributed Systems* 13 (1), 14–25, 2002.

[28] H. Takagi and L. Kleinrock, "Optimal transmission ranges for randomly distributed packet radio terminals." *IEEE Transactions on Communications* 32 (3), 246-257, 1984.

[29] I. Stojmenovic, M. Russell, and B. Vukojevic, “Depth first search and location based localized routing and QoS routing in wireless networks.” *Computers and Informatics* 21 (2), 149-165, 2002.

[30] G. Chelius, E. Fleury, and A. Busson. “From Euclidian to hop distance in multi-hop radio networks: a discrete approach.” Tech. Report INRIA RR-5505, feb. 2005.

[31] M. Woo S. Singh, and C. Raghavendra, “Power-aware Routing in Mobile Ad-hoc Networks.” In *Proc.4th ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom’98)*, (1998), pp. 181–190.

[32] J. Kuruvila, A. Nayak, and I. Stojmenovic, “Progress based localized power and cost aware routing algorithms for ad hoc and sensor wireless networks.” *International Journal of Distributed Sensor Networks*. to appear.

[33] I. Solis, and K. Obraczka, “The impact of timing in data aggregation for sensor networks.” In *Proc. of the IEEE International Conference on Communications (ICC 2004)*, (Paris, France, 2004).

[34] R. Cristescu, B. Beferull-Lozano, and M. Vetterli, “On network correlated data gathering.” In *Proc. of INFOCOM’04*, pp. 285–293.

[35] J. Wieselthier, G. Nguyen, and A. Ephremides, “On the construction of energy-efficient broadcast and multicast trees in wireless networks.” In: *Proc. INFOCOM’2000*, (Tel Aviv, Israel, 2000), pp. 585–594.

[36] N. Li, J. Hou, and L. Sha, “Design and analysis of an MST based topology control algorithm.” In *Proc. INFOCOM’2003*, (San Francisco, CA, USA, 2003).

[37] C. Intanagonwiwat, R. Govindan, and D. Estrin, “Directed diffusion: a scalable and robust communication paradigm for sensor networks.” In *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom’2000)*, (Boston, MA, USA, 2000), pp. 56–67.

[38] J. Carle, and D. Simplot-Ryl. “Energy efficient area monitoring by sensor networks.” *IEEE Computer* 37 (2), 40-46, 2004.

[39] D. Simplot-Ryl, I. Stojmenovic, and J. Wu, “Energy Efficient Backbone Construction, Broadcasting, and Area Coverage in Sensor Networks.” Chapter in *Handbook of Sensor Networks: Algorithms and Architectures*, I. Stojmenovic (ed.), Wiley & Sons, 2005.

[40] S. Fikret, and Y. Bulent, “Time Synchronization in sensor Networks: a survey.” *IEEE Network* 18 (4), 2004, 45-50.

[41] A. M. Naimi, and P. Jacquet. One-Hop Delay Estimation in 802.11 Ad Hoc Networks Using the OLSR Protocol. Tech. Report INRIA RR-5327, oct. 2004.

Les réseaux de terrain : des réseaux temps réel

Jean-Pierre Thomesse
LORIA - Institut National Polytechnique de Lorraine
2, avenue de la forêt de Haye
54516 Vandœuvre-lès-Nancy
Jean-Pierre.Thomesse@ensem.inpl-nancy.fr

Résumé

Les réseaux de terrain constituent une race particulière de réseaux. Réseaux parmi les plus divers par la nature des stations connectées, par les applications concernées, par les protocoles utilisés, par le nombre de normes, ils sont certainement les réseaux qui supportent le plus de contraintes, contraintes d'environnement, de sûreté de fonctionnement, de temps réel, de besoin de dynamique, voire de « plug and play ».

Longtemps cantonnés dans le milieu industriel, ils ont colonisé le monde des transports, des systèmes embarqués, des mobiles, de la domotique, et seront à la base des systèmes multi - agents du futur.

Les besoins en termes de services sont relativement standards, mais les qualités de service requises dépendent des applications, et évidemment des protocoles utilisés.

1. Introduction

Les réseaux de terrain connectent les capteurs, les actionneurs, les contrôleurs dans n'importe quel système automatisé, dans l'industrie, dans le bâtiment ou les transports. Ce sont des réseaux particuliers qui relèvent à la fois des réseaux industriels, des réseaux embarqués et des réseaux temps réel. Ce sont ces derniers aspects que nous allons présenter ici.

Les réseaux de terrain sont nés au début des années 1980 après quelques expériences de systèmes d'acquisition de données ou de réseaux d'automates programmables qui avaient vu le jour durant la décennie précédente, dans le monde industriel ou de la recherche en physique nucléaire. Les principaux réseaux (préhistoriques ?) sont les réseaux CAMAC [1], ARCNET [2] et [3], des réseaux de constructeurs [4], comme Data Highway d'Allen Bradley et Tiway de Texas Instruments [5] et [6] et un réseau utilisé dans l'avionique militaire (MIL Std 1553 [7]). Deux réseaux ont laissé leurs noms à la postérité, le réseau MODBUS de la société Modicon [8] et [9], et WDPF de la société Westinghouse [10].

Les véritables réseaux de terrain actuels voient le jour au début des années 1980, car plusieurs conditions sont alors réunies que nous analyserons brièvement au paragraphe 2.

Alors en 1985, commence la saga de la normalisation qui n'est toujours pas terminée. Les cahiers des charges différaient selon les promoteurs et les domaines d'applications s'étant étendus au cours de ces années, il ne faut pas s'étonner du nombre de normes toutes évidemment incompatibles. Le paragraphe 3 résume ces péripéties et présente les principales normes actuelles.

Les paragraphes 4 et 5 présenteront des taxonomies des réseaux de terrain selon les principes des protocoles de Medium Access Control et les modèles de coopération de la couche application. Des considérations sur les architectures à plusieurs piles de protocoles compléteront ce chapitre avant de conclure sur quelques perspectives ouvertes.

2. L'apparition des réseaux de terrain

Les réseaux se développent dans tous les domaines, les réseaux publics, les réseaux locaux dans les bureaux et l'industrie. La fin des années 70s est marquée par la rivalité entre les techniques « Ethernet » [14] et « jeton » [13]. C'est le moment où les utilisateurs réclament plus de normes, plus de compatibilités entre produits de constructeurs différents. C'est ainsi que General Motors lance le projet Manufacturing Automation Protocol (MAP) [17] pour diminuer les coûts d'automatisation de ses usines. Dans le même temps, Boeing lance le projet Technical and Office Protocol (TOP) [16] avec un objectif similaire de faciliter les relations entre bureaux d'études et usines en normalisant les systèmes de communication [15]. Ces projets viennent compléter les modèles d'architecture des systèmes automatisés, tels que le modèle Computer Integrated Manufacturing (CIM) du NBS (National Bureau of Standards) [18].

Ce modèle met en évidence l'absence de normes pour couvrir les besoins de communication entre les instruments et les dispositifs de terrain. La demande

industrielle est aussi appuyée pour simplifier le câblage traditionnel en point à point des entrées – sorties.

Pour répondre à ces besoins, 1980 voit la première version du modèle de référence Open System Interconnection (OSI) [11], [12] de l'ISO (International Organization for Standardization). La microélectronique est en plein développement ; les idées d'intégration de la communication à faible coût dans des constituants bon marché deviennent réalistes, les capteurs et actionneurs intelligents vont pouvoir voir le jour. Un rapport du professeur Soutif [24] préconisait le développement de transmissions numériques pour développer l'industrie des instruments de mesure. Un colloque au Royaume Uni [25] tirait des conclusions semblables. Toutes les conditions sont réunies pour innover (besoin industriel, technologies disponibles), ainsi vont naître les réseaux de terrain, le mot lui-même (fieldbus en anglais) apparaîtra en 1985.

3. Les besoins et la normalisation

C'est en 1985 que commence la normalisation au sein du comité technique TC65/SC65C/WG6 de l'International Electrotechnical Commission (IEC). Plusieurs projets de réseaux de terrain sont alors à l'étude en Europe essentiellement.

3.1. Les premiers travaux

FIP (Factory Instrumentation Protocol) qui deviendra WorldFIP, est spécifié en France entre 1982 et 1984 [19] et [20]. PROFIBUS (Process Field Bus) [21] naît en Allemagne en 1984. P-Net [22] démarre au Danemark en 1983, ainsi que CAN (Controller Area Network) [23] en Allemagne par la société Bosch. FIP et PROFIBUS sont définis par des consortiums d'industriels et de laboratoires, avec toutefois une différence importante : le consortium initial FIP ne contient pas de représentants d'offreurs, alors que le consortium PROFIBUS ne contient pas de représentants d'utilisateurs. Ce qui expliquera que le réseau FIP proposera le premier (et sera longtemps le seul) des services ou des propriétés (comme la cohérence) tout à fait innovants sur lesquels on reviendra plus tard (cf. §XX).

3.2. Le cahier des charges

3.2.1. La synthèse

Ces travaux en Europe et d'autres aux USA, comme ceux sur le réseau HART (Highway Addressable Remote Transducer) [26] par la société Rosemount conduisaient l'IEC et l'ISA (Instrumentation Society of America) à mener en commun les travaux de normalisation. Ces travaux commencèrent par l'établissement d'un cahier des charges selon un modèle déductif d'établissement de spécification et un questionnaire [27]. Les besoins étaient donc relevés et synthétisés, conduisant à de nombreuses versions dans les années 1986 et 1987. ils sont résumés dans le tableau 1.

3.2.2. Les services

En termes de services, le cahier des charges met en évidence les opérations de lecture et d'écriture de variables, ou d'objets. D'autres services sont évoqués pour les opérations de maintenance, de configuration et plus généralement de gestion de réseau. D'autres services non explicités sont également évoqués au travers de certains aspects protocolaires (services sans acquittement, services de diffusion).

Ces services seront complétés plus tard vers la fin des années 1980, grâce à l'émergence du protocole MMS (Manufacturing Message Specification) [34] issu du projet MAP.

3.2.3. La qualité de service

On remarquera que ce que l'on appelle aujourd'hui la qualité de service apparaît dans quelques rubriques :

- l'intégrité des données mais rien n'est exigé quant à l'ordre des messages,
- le temps de réponse dont la définition varie ; pour l'IEC c'est le délai entre l'occurrence d'un événement et sa signalisation, pour l'ISA c'est le délai entre une requête et la délivrance de l'information,
- les débits en termes de fréquence d'échantillonnage, et de vitesse de rafraîchissement,
- la cohérence des données, cohérence temporelle et consistance (aussi appelée cohérence spatiale).

Ce cahier des charges met implicitement en évidence les deux types de trafic, périodique et apériodique, issus respectivement de la théorie des systèmes échantillonnés, et des besoins de configuration et maintenance des stations.

Ce sont essentiellement les moyens (donc les mécanismes protocolaires) de satisfaire ces deux types de besoins qui vont permettre de distinguer et classifier les réseaux de terrain. Ce sont eux aussi qui détermineront la qualité de service de ces divers réseaux.

3.3. Les normes

Les deux comités de l'IEC et l'ISA vont dès 1987 tenter de trouver un compromis pour ce qui devait être LA NORME INTERNATIONALE. Peine perdue, le seul succès fut la norme de couche physique [30]. Le sujet fit les beaux jours des journaux du domaine, des congrès et salons, sans le moindre succès, au grand dam de nombre de spécialistes dont Patricio Leviti s'est fait l'interprète dans un article célèbre [31]. Le lecteur pourra aussi se référer à un article récent de l'auteur [28] pour plus de détail sur ces épisodes.

TABLE 1. CAHIER DES CHARGES

	IEC	ISA	WorldFIP
Application	Format d'APDU Pour le process Mesures Alarmes Etats Etat des éléments terminaux Pour l'installation Numéro des éléments Information pour la maintenance Données du fabricant	Format d'APDU Différents types de données Binaires (16 bits) Réels (32 bits) Autres optionnels Echange de données avec état Messages non sollicités Données de Configuration Etat des équipements Contrôle d'accès avec mot de passe	Fonctions requises Deux types d'échanges: Données identifiées Messages Complète connectivité (tout processus a accès à toute donnée)

	IEC	ISA	WorldFIP
Application architecture Propriétés de Cohérences			Distribuée ou centralisée Synchrone ou asynchrone Cohérence temporelle des données et des actions Consistance des copies
Isolation	Isolation galvanique à 500V	Isolation galvanique à 250V	
Alimentation	Alimentation possible par le réseau	Alimentation possible à partir de la boîte de jonction	Stations alimentées par le réseau
Atmosphère inflammable	Spécificités pour connexions en environnements dangereux	Sécurité intrinsèque optionnelle	Sécurité intrinsèque possible
Medium	Paire torsadée industrielle Réutilisation de câblage Coaxial	Paire torsadée industrielle Réutilisation de paire torsadée blindée	Paire torsadée blindée Coaxial et fibre optique Réutilisation de câblage
Topologie	Simple bus avec « bretelle » ou avec « déviation »	Simple bus avec étoile à chaque boîte de jonction	
Gestion des Stations	Addition et retrait possible de capteurs et contrôleurs en ligne.		
Configuration	Modalités de configuration définies dans la norme	Modalités de configuration définies dans la norme	
Redondance		Media	Media, tous composants
Performances			
Nombre de stations	30 stations	5 à 32 pour H1, 32 pour H2	
Longueur et débits	350m –150 mess / seconde 350 m – 10 000 mess/seconde 40 m – 5 000 mess/seconde	Deux distances : 750 m et 1850 m H1: 10 messages par équipement et par seconde	Fréquence d'échantillonnage selon les applications
Temps de réponse	5 ms pour le manufacturier 20 ms pour le continu	H1: 100ms H2: 1 ms	
Intégrité des données		Une erreur non détectée tous les 20 ans	
Adressage	D'une station et de chaque élément final d'une station	Maximum 256 adresses	Nombre maximum d'éléments transmis
Protocoles	Echange entre stations ou entre éléments terminaux Avec ou sans retransmission Transfert du contrôle d'accès	READ/WRITE des messages sur les équipements	Maître simple ou multiple, Broadcast, Multicast , avec ou sans retransmission

TABLE 2. IEC 61158 ET IEC 61784-1

IEC 61784 Profiles	IEC 61158				Noms des réseaux
	Couche physique	Couche Liaison	Couches Transport Réseau	Couche Application	
CPF-1/1	Type 1	Type 1		Type 9	Foundation Fieldbus H1
CPF-1/2	Ethernet	Ethernet	TCP/UDP/IP	Type 5	Foundation Fieldbus HSE
CPF-1/3	Type 1	Type 1		Type 9	Foundation Fieldbus H2
CPF-2/1	Type 2	Type 2		Type 2	ControlNet
CPF-2/2	Ethernet	Ethernet	TCP/UDP/IP	Type 2	EtherNet/IP
CPF-3/1	Type 3	Type 3		Type 3	Profibus-DP
CPF-3/2	Type 1	Type 3		Type 3	Profibus-PA
CPF-3/3	Ethernet	Ethernet	TCP/UDP/IP	Type 10	PROFInet
CPF-4/1	Type 4	Type 4		Type 4	P-Net RS 485
CPF-4/2	Type 4	Type 4		Type 4	P-Net RS 232
CPF-5/1	Type1	Type 7		Type 7	WorldFIP (pile complète)
CPF-5/2	Type1	Type 7		Type 7	WorldFIP (DWF)
CPF-5/3	Type1	Type 7		Type 7	WorldFIP (profil interm)
CPF-6/1	Type 8	Type 8		Type 8	Interbus (pile initiale)
CPF-6/2	Type 8	Type 8	TCP/IP	Type 8	Interbus (sur IP)
CPF-6/3	Type 8	Type 8		Type 8	Interbus
CPF-7/1	Type 6	Type 6			SwiftNet (réduit)
CPF-7/2	Type 6	Type 6		Type 6	SwiftNet (complet)

Le résultat est aujourd'hui un ensemble de profils tous incompatibles, une caricature de ce que doit être une norme.

3.3.1. IEC 61158

IEC 61158-3 et 4 pour les spécifications de services et les protocoles de couche liaison de données, qui couvrent 8 types de réseaux.

Type 1 est l'ancien projet TR1158, le compromis obtenu techniquement mais refusé par une minorité de pays en 1999.

Type 2 est la spécification ControlNet,

Type 3 est la spécification PROFIBUS,

Type 4 est la spécification P-Net,

Type 5 est la spécification FOUNDATION Fieldbus,

Type 6 est la spécification SwiftNet,

Type 7 est la spécification WorldFIP,

Type 8 est la spécification INTERBUS.

IEC 61158-5 et 6 pour les spécifications de services et les protocoles de couche application qui couvrent 10 types de réseaux.

Les huit premiers correspondent à ceux de la couche liaison de données. Les deux autres Type 9 et Type 10, définissent le réseau H1 de FOUNDATION Fieldbus et le réseau PROFInet.

Deux autres parties, 61158-7 pour la gestion de réseau et 61158-8les tests de conformité ont été annulées, à cause de l'existence de nombreux outils propriétaires. La maintenance de cette norme est confiée à un sous-groupe SC65C/MT1, (MT signifie Maintenance Team).

3.3.2. IEC 61784

Ce projet de norme tente d'uniformiser la présentation de la précédente en simplifiant les profils.

Le tableau 2 présente la situation actuelle de la partie 1 de cette norme. Il est en partie issu de [32].

La partie 2 de cette norme est dédiée à tous les réseaux de terrain qui utilisent Ethernet comme protocole de Medium Access Control. C'est le projet intitulé « Digital data communications for measurement and control – Part2 Additional profiles for ISO/IEC 8802.3 based communication networks in real time applications ».

Deux autres parties du projet de norme 61784 concernent les aspects sûreté et sécurité des réseaux. Ce sont les parties 3 et 4 respectivement intitulées:

« *Profiles for functional safe communications in industrial networks* » et

« *Profiles for secure communications in industrial networks* ».

CPF signifie Communication Profile Family, et RTE Real Time Ethernet) :

La partie deux se compose des propositions suivantes :

TABLE 3. IEC 61158 et IEC 61784-1

CPF 2	ControlNet - RTE
CPF 3	Profibus - Profinet- RTE
CPF 4	P-Net- RTE
CPF 6	Interbus- RTE
CPF 10	Vnet/IP- RTE
CPF 11	TCnet- RTE
CPF 12	EtherCAT- RTE
CPF 13	ETHERNET Powerlink- RTE
CPF 14	EPA- RTE
CPF 15	MODBUS-RTPS - RTE
CPF 16	SERCOS - RTE

4. Les principaux protocoles MAC

Les réseaux de terrain doivent satisfaire un trafic périodique et un trafic apériodique. La qualité de service du premier type s'exprime en respect des fréquences sans gigue de rafraîchissement des variables applicatives, considérant la période comme la durée de vie de l'information. Celle du second s'exprime en termes de temps de réponse, i.e. le délai entre la demande d'émission et l'instant de transmission. Indépendamment de l'aspect temporel, certains réseaux de terrain distinguent deux types de trafic, celui des couples (nom d'un objet, valeur et statut de l'objet) et tous les autres messages associés à n'importe quelle application. Le premier trafic est similaire à celui qui a été retenu par l'IEEE dans la norme LLC type 3, et le second est celui qui est naturellement considéré dans le modèle OSI et toutes les communications qui en découlent.

4.1. Le trafic périodique

Le trafic périodique est géré de façon centralisée par des techniques de TDMA ou par scrutation. Les réseaux suivants fonctionnent selon TDMA: TTP-A, SERCOS, ARINC, INTERBUS, ControlNet. Les réseaux WorldFIP, PROFIBUS-DP et PA, FOUNDATION Fieldbus et P-NET sont régis par un protocole de scrutation. Le trafic périodique géré de façon décentralisée l'est, soit par jeton (PROFIBUS FMS et P-NET), soit par contention selon CSMA-CD ou ses variantes (CAN, LonWorks, DeviceNet, SDS, Batibus).

4.2. Le trafic apériodique

Le trafic apériodique est géré lui aussi de façon soit centralisée, soit décentralisée. Dans le cas décentralisé, une technique de jeton est utilisée dans le cas de ControlNet, de P-Net, et de PROFIBUS. Les réseaux fonctionnant selon CSMA l'appliquent aussi bien pour l'apériodique que pour le périodique, la priorité de l'un sur l'autre est une décision locale à chaque station. Pour le cas centralisé, le trafic apériodique est forcément géré par un serveur périodique, et deux techniques prévalent, celle d'un champ laissé libre dans chaque trame périodique (INTERBUS) et celle de trame spéciale sur demande (PROFIBUS-DP, WorldFIP, FOUNDATION Fieldbus).

4.3. La qualité de service

La qualité de service est en général exprimée selon différentes caractéristiques relatives à l'intégrité des données, au transport, à la sécurité, à la sûreté de fonctionnement, aux capacités, aux aspects temporels et aux cohérences [33]. Ces divers aspects sont différemment traités dans les diverses normes évoquées au paragraphe précédent. Dans la norme IEC 61168, la qualité de service intègre la présence ou non des mécanismes suivants : communication avec ou sans connexion, stratégie d'ordonnancement, méthode de stockage des PDU (files d'attente ou buffers), différentiation de trafic, ordre des PDU et attributs temporels.

Précisons quelques unes de ces dispositions.

4.3.1. Buffers ou files d'attente

Dans le modèle OSI, les demandes de service émanant d'une couche N vers une couche N-1 sont normalement traitées l'une après l'autre avec ou sans priorité. Ici, dans le cas des demandes d'émission périodique de valeurs de variables, ce qui importe est plus souvent de transmettre la valeur la plus récente, plutôt que toutes les valeurs systématiquement dans leur ordre d'arrivée. C'est ainsi que plusieurs réseaux fonctionnent (WorldFIP, CAN, Foundation fieldbus, INTERBUS).

4.3.2. Attributs temporels

La transmission d'une valeur entre deux buffers doit être sécurisée, par exemple en indiquant si une valeur transmise a été rafraîchie depuis la transmission précédente. Un système de datation peut être utilisé , mais aussi des attributs temporels pour qualifier une valeur. C'est ainsi que trois attributs booléens : résidence, mise à jour, synchrone ont été introduits pour qualifier de quand date la dernière mise à jour ou la lecture d'une unité de donnée.

L'attribut de résidence indique que la durée de la présence de la valeur d'une unité de donnée est inférieure à une borne donnée.

L'attribut de mise à jour indique que la production de la valeur d'une unité de donnée a eu lieu dans un intervalle de temps défini par un événement d'origine et une durée.

L'attribut synchrone indique si une mise à jour et une lecture d'une unité de donnée ont eu lieu dans un intervalle de temps donné entre un événement d'origine et une durée.

Ces attributs présents dans le type 1 de la norme proviennent en fait de la norme française du réseau WorldFIP.

5. Les modèles de coopération

Les modèles de coopération représentent la façon dont les processus d'application peuvent coopérer pour atteindre l'objectif commun. Ils relèvent de la famille du modèle client – serveur ou du modèle producteur – consommateur.

5.1. Le modèle client – serveur

Le modèle client – serveur définit la façon dont un service est accessible par les processus qui en ont besoin. Il est défini par les quatre primitives de demande (request), d'indication, de réponse et de confirmation. Il est temporellement caractérisé par deux délais : le délai local au serveur entre la réception d'une indication et la réponse, et le délai de réponse global local au client entre la requête et la confirmation. Il suppose deux échanges : le transfert de la demande, et celui de la réponse.

Ce modèle très ancien est bien adapté quand un service est occasionnellement requis, mais il est évident que dans le cas d'échanges périodiques, il peut être simplifié. Par ailleurs il ne répond pas aux besoins de coordonner plus de deux processus. Par contre il permet de définir n'importe quel service. L'exemple type de ce modèle est MMS (Manufacturing Message Specification) [34] qui est à la base du protocole FMS (FieldBus Message Specification) de Profibus qui a inspiré de nombreux autres protocoles.

5.2. Le modèle producteur - consommateur

Le modèle producteur consommateur repose sur l'idée que le producteur d'une information est responsable de son transfert vers celui qui en an besoin. C'est un modèle qui offre deux primitives services émettre et recevoir. Il est à la base du protocole MPS de WorldFIP. C'est aussi le protocole des réseaux basés sur CAN, comme DeviceNet et SDS. Il est également présent dans MMS au travers de trois services InformationReport (envoi d'une donnée), EventNotification (envoi d'un événement), UnSollicitedStatus (envoi d'un état). Ce modèle ne permet pas de demander n'importe quel service sauf à construire une interprétation des informations produites et consommées au-dessus d'un tel protocole. Ce modèle est aussi appelé « Publisher – Subscriber ». Etudions maintenant les modèles multipoint.

5.3. Le modèle producteur - consommateurs

Ce modèle s'appuie sur le précédent avec une hypothèse de mécanisme de diffusion sous-jacent. Ainsi plusieurs consommateurs peuvent espérer recevoir l'information émise par un producteur. Deux sous - modèles ont été introduits par la norme 61158, le « push-model » et le « pull-model ».

5.3.1. Le push-model

Le push-model est celui qui a été défini sur le principe ci-dessus, à savoir que le producteur « pousse » l'information vers les consommateurs. Ceux-ci s'abonnent auprès d'un producteur par le biais d'un service sur le modèle client-serveur.

5.3.2. Le pull-model

Le pull-model repose dur l'existence d'un abonné consommateur particulier qui joue le rôle d'un client vis-à-vis du producteur (serveur) dont la réponse est diffusée à tous les consommateurs.

C'est une façon de réaliser un modèle multiclients – serveur.

5.3.3. Propriété de cohérence temporelle

On appelle cohérence temporelle la propriété de plusieurs événements de s'être produits dans une fenêtre temporelle donnée. On définit ainsi la simultanéité. Si on fait l'hypothèse d'une diffusion sans erreur, aux délais de propagation près, les réceptions de l'information chez les consommateurs sont des événements simultanés ou cohérents temporellement. On peut ainsi commander des actions considérées comme simultanées, par exemple des acquisitions réparties de données, qui pourront elles aussi être considérées comme simultanées.

Cette propriété est supposée vérifiée quand les transmissions sont sans erreur et que les attributs temporels sont corrects. Des unités de données sont alors dites temporellement cohérentes du point de vue de leur acquisition quand elles ont été acquises simultanément. On peut définir la même propriété vis-à-vis de la transmission, et de la réception.

5.4. Le modèle Producteurs - Consommateurs

Ce modèle est une extension du précédent dans le sens où on considère comme une seule transaction une suite de plusieurs transactions élémentaires du type précédent. C'est pour définir une propriété de cohérence sur plusieurs échanges que ce modèle a été introduit. On considère une liste d'unités de données en provenance de plusieurs producteurs et copiées chez plusieurs consommateurs. La propriété de cohérence spatiale d'une liste d'unités de données est vraie quand les copies de cette liste sont identiques sur plusieurs sites consommateurs. Le mécanisme de vérification de cette propriété s'apparente à un acquittement global à l'égard de plusieurs émetteurs.

5.5. Le modèle client - multiserveurs

Le modèle client – multi serveur est un cas particulier dans lequel la requête est traitée par plusieurs serveurs partiels. La décomposition de la requête est a priori inconnue du client. Les cas d'erreurs doivent être étudiés.

Un cas particulier est le modèle « 3ème partie » dans lequel, le service initial normalement fourni par le serveur est sous-traité à une troisième partie. Plusieurs cas sont intéressants à étudier.

5.6. Conclusion

Tous ces modèles ne sont pas délivrés par tous les réseaux de terrain. Il est évident que la présence de tel ou tel modèle facilitera une certaine distribution des processus sur plusieurs sites. Les aspects temps réel sont plus ou moins facilement gérés selon la présence ou non des mécanismes et attributs vus dans les paragraphes ci-dessus.

6. Les architectures

Les architectures des réseaux de terrain sont très diverses, depuis les protocoles de couche physique (que nous n'avons pas étudiés) jusqu'aux protocoles de couche application, en passant par les protocoles de couche MAC. Ces réseaux doivent par ailleurs s'interfacer avec les autres réseaux de l'entreprise et ainsi participer à l'intégration des différentes fonctions de l'entreprise. Il est alors nécessaire de partager certains services et protocoles entre les réseaux de terrain et les autres essentiellement basés sur les protocoles de l'internet.

La plupart des réseaux ont traité ce problème en offrant plusieurs piles de communication, l'une pour le temps réel et les services relevant du ccontrôle-commande, l'autre pour les opérations à distance, incluant la gestion des équipements au travers de pages Web.

Plusieurs architectures ont été ainsi étudiées, et l'un des enjeux des prochaines années est certainement la définition d'un modèle unique d'architecture des réseaux , couvrant l'ensemble des besoins.

Une architecture à trois piles proposée par T. Phinney [35] pourrait être réétudiée à l'aune des nouvelles propositions de normes basées sur Ethernet. En supposant exister un protocole de couche liaison avec des propriétés temps réel, on peut construire sur cette base trois piles de protocole, l'une avec uniquement les couches application et présentation spéciales temps réel, la deuxième avec les couches applications et présentation d'intérêt général sans routage ni transport pour des applications très locales, et la troisième pile avec les mêmes couches d'application et présentation , mais sur les couches normales, réseau et transport.

7. Conclusion

Ce chapitre a dressé un panorama des réseaux de terrain, en montrant leurs caractéristiques temps réel. Il faut noter que le principal moyen de piloter les aspects temps réel, à savoir l'ordonnancement , n'est pas partie prenante des protocoles de ces réseaux, excepté pour le réseau CAN qui de façon native, possède un ordonnancement par priorité. L'ordonnancement est fixé par les utilisateurs, en dehors de toute norme. C'est vrai aussi bien dans l'établissement des tables de scrutation, dans la gestion dynamique des demandes de trafic apériodique, dans la définition des slots de temps, ou dans la hiérarchisation des messages selon les techniques protocolaires utilisées.

Quelques problèmes ouverts ont été évoqués. Les projets de normes basés sur Ethernet constituent une mine de sujets soit d'évaluations, de comparaisons, d'intégration de mécanismes temps réel, etc. Notons toutefois que l'usage du mot Ethernet est parfois très abusif. Il ne reste parfois digne du nom, que le format des PDU ! On n'est plus du tout dans le cas du CSMA-CD, mais dans des mécanismes de commutation, parfois de scrutation centralisée, avec des liaisons full-duplex.

Quelques problèmes nouveaux apparaissent avec les réseaux mobiles de capteurs (reconnaissance de l'environnement, routages dynamiques…)

Les méthodes, mécanismes et outils de distribution sont toujours d'actualité, surtout avec des objectifs de validation.

Enfin à la limite du monde du génie logiciel et de celui des réseaux, citons les travaux sur les « functions blocks », sorte de composants, dont les modèles et les spécifications sont encore loin de satisfaire les besoins d'ouverture et d'interopérabilité.

8. Références

[1] [L. Costrell, "CAMAC instrumentation system - introduction and general description". IEEE-Transactions-on-Nuclear-Science. April 1971; ns-18(2), pp. 3-8.

[2] R. P. Carson, "Distributed data analysis in computer networks". Industrial-Research/Development. May 1981; 23(5): 130-5.

[3] J. A. Murphy, "Token-passing protocol boosts throughput in local networks". Electronics. 8 Sept. 1982; 55(18), pp. 158-163.

[4] J. F. Peyrucat, "Interautomaton communication via networks". Mesures,-Regulation,-Automatisme. 24 Jan. 1983; 48(1), pp. 35-37, 39, 41.

[5] C. W. Rose and J. D. Schoeffler, Microcomputers in instrumentation and data acquisition systems, Analysis-instrumentation,-vol.12. 1974, pp. 157-163.

[6] P. H. Troutman, "A digital link for controllers and valves". Instrumentation-Technology. July 1978; 25(7), pp. 55-7.

[7] Gifford,-C.-A., (1974). “A military standard for multiplex data bus”. Proceedings of the IEEE-1974, National Aerospace and Electronics Conference. 13-15 May 1974 Dayton, OH, USA, pp. 85-8.
[8] E. Hassler, “Industrial master/slave system”. Technische-Rundschau. 8 April 1980; 72(14), pp.17-18.
[9] E. Hassler, “An industrial master-slave system. Organisation of communications for decentralized freely-programmable controls”. Technische-Rundschau. 29 April 1980; 72(17), pp. 25-6.
[10] T. H. Schwalenstocker, “A process control system using multibus”. Proceedings of the First Annual Control Engineering Conference. 18-20 May 1982 Rosemont, IL, USA 1982, pp. 133-137.
[11] ISO, ISO 7498, Data Communications, Open System Interconnection – Basic Reference Model.
[12] H. Zimmermann, “OSI reference model. The ISO model of architecture for open system interconnection”, IEEE Trans. COM-28 n°4, April 80, pp. 425-432.
[13] ISO, Information processing systems, Local area networks-Part4: Token Bus access method. 1990.
[14] [14] ISO, Information processing systems, Local area networks-Part3: Carrier Sense Multiple Access-Collision Detection. 1990.
[15] S. R. Dillon, “Manufacturing automation protocol and technical and office protocols - success through the OSI model”. COMPCON-Spring-'87.-Thirty-Second-IEEE-Computer-Society-International-Conference. 23-27 Feb. 1987 San Francisco, CA, USA, pp. 80-81.
[16] N. Collins, “Boeing architecture and TOP (technical and office protocol)”. Networking:-A-Large-Organization-Perspective. April 1986, Melbourne, FL, USA, pp. 49-54.
[17] R.-D Floyd. “Manufacturing Automation Protocol”. In IEEE International Conference on Communications, 1985, 23, 26 June. Chicago, IL, USA. pp 620-624.
[18] P. F. Brown and C. R. Mac Lean, “The architecture of the NBS factory automation”. IFAC Congress, Münich, Germany. 1987.
[19] D. Galara and J.-P. Thomesse, Groupe de réflexion FIP. Proposition d'un système de transmission série multiplexée pour les échanges d'informations entre des capteurs, des actionneurs et des automates réflexes. Ministère de l'Industrie et de la Recherche. Mai 1984, 56 pages. Published also by Editions KIRK, Paris, France, 1991.
[20] J.-P. Thomesse, “Le réseau de terrain FIP”, Revue Réseaux et Informatique Répartie, Hermès, Vol 3, N°3, 1993, pp. 287-321.
[21] DIN, German Standards 19245-1 to 19245-3, PROFIBUS, Process fieldbus. 1990.
[22] DS, Danish Standard, DS 21906. P-Net, Multi-master, multi-net fieldbus for sensor, actuator and controller communications.
[23] CAN, Bosch CAN Specification-Version 2.0 Part A, R. Bosch Gmbh, Germany, 1991.
[24] M. Soutif, Rapport sur l'Industrie des Instruments de Mesure, Ministère de la recherche, 1982, Paris, France.
[25] IEE, Colloquium on `Distributed Process Control; Today and Tomorrow', IEE, 1 March 1982 London, UK
[26] S. McClelland, “A Hart to Hart with Rosemount”. Sensor-Review. April 1989; 9(2): 71-74.
[27] ISA, SP50 “Field bus standard for use in industrial control systems”, “discussion draft and questionnaire for functional requirements”. 1986.
[28] J.-P. Thomesse, “Fieldbus Technology in Industrial Automation”. Proceedings of the IEEE, June 2005, vol 93, N°6, pp.
[29] G. G. Wood, “Current fieldbus activities”. Computer Communications. Vol 11, N°3, pp 118-123.
[30] IEC, IEC Standard 1158-2, Fieldbus standard for use in industrial control systems- Part 2 Physical layer specification and service definition + AMD1 (1995).
[31] P. Leviti, “IEC 61158: an offence to technicians”. IFAC Int. Conf. on Fieldbus Systems and their Applications, FET’2001. Nov 15-16 2001. Nancy, France, Ed by Dietrich, Neumann and Thomesse, Pergamon, pp 9-16.
[32] T.Sauter, Fieldbus Systems: History and Evolution. In The Industrial Communication technology, CRC Press, 2005, R. Zurawski Ed. pp 7-1, 7-39.
[33] ISO, DIS 13236 Information technology-Quality of Service-Framework. 1996.
[34] ISO, ISO/IEC IS 9506 Manufacturing Message Specification. 1990.
[35] T. P. Phinney, D. Brett, D. McGowan and Y. Kumeda, “FieldBus-Real-Time comes to OSI”. 10th Annual International Phoenix Conference on Computers and Communications, IEEE Comput Soc. Press, Los Alamitos, CA, USA, 1991, pp. 594-599.

Approches (m,k)-firm pour la gestion de la qualité de service temps réel

YeQiong SONG
LORIA - INPL
Campus Scientifique, B.P. 239
54506 Vandoeuvre-Lès-Nancy, France
song@loria.fr

Résumé

Cet article présente d'abord un état de l'art sur les principaux algorithmes d'ordonnancement développés pour la garantie temps réel (m,k)-firm, puis explique comment les approches (m,k)-firm peuvent être utiles pour une meilleure gestion de la qualité de service avec dégradation contrôlée (graceful degradation) dans les réseaux et systèmes temps réel. Un algorithme appelé (m,k)-WFQ est détaillé pour illustrer l'intérêt de l'approche (m,k)-firm dans l'ordonnancement des paquets de flux MPEG dans des réseaux. Le problème fondamental de la garantie déterministe de (m,k)-firm est également approfondi à travers l'élaboration de la condition suffisante d'ordonnançabilité de l'algorithme DBP.

1. Introduction

Aujourd'hui, de plus en plus d'applications temps réel sont déployées au-dessus de réseaux comme l'Internet. Ceci signifie que ces réseaux doivent fournir des garanties en termes de respect de contraintes temporelles sur les communications. L'augmentation constante de débit des réseaux qui composent l'Internet (Ethernet, réseaux mobiles sans fil, réseaux courants porteurs par exemple) n'apporte une solution que temporaire. En effet, le surplus de bande passante apportée par toute augmentation de débit est pratiquement immédiatement comblée par de nouvelles applications multimédias. La technique de réservation qui consiste à sur-dimensionner les ressources (over-provisioning) n'est, donc, pas une solution viable à l'avenir. Ce constat oblige donc à spécifier des techniques et méthodes pour garantir une Qualité de Service (QdS) temporelle sous des contraintes de ressources limitées. De plus, la gestion de la QdS temporelle dans l'Internet soulève un problème de passage à l'échelle. Pour y faire face, l'IETF recommande d'appliquer l'architecture « Diffserv » plutôt que « Intserv ». Mais le problème de « Diffserv » est que la garantie est vis à vis de classes de trafics et non pour une application. Une exigence de garantie déterministe de QdS pour une application dans une classe d'applications oblige à un dimensionnement selon la contrainte la plus stricte imposée aux applications considérées, conduisant de nouveau à un surdimensionnement de ressources. Notons que des solutions pour supporter des applications sous contraintes temps réel souples sur l'Internet ont été proposées [El-Gendy03]. Mais ces solutions n'apportent qu'une garantie temps réel probabiliste. Ceci peut ne pas convenir à certains types d'applications, notamment dans le cas du contrôle-commande ou du multimédia. En effet, ces applications peuvent tolérer des pertes de paquets (ou des paquets écartés à cause du retard dépassant l'échéance requise) en transmission sur les réseaux mais à condition que ces pertes soient distribuées selon un modèle spécifié de manière déterministe et non en observant une propriété sur la moyenne de leurs occurrences (par exemple, transmission de paquet de MPEG, de voix sur IP, …). Un problème typique est comment éviter trop de pertes consécutives des paquets en cas de congestion des réseaux. Il est clair que les politiques classiques de gestion de buffers telles que TD (Tail-Drop) conduit inévitablement à des pertes consécutives tandis que RED (Random Early Detection), en écartant aléatoirement des paquets dans une région de taille de file d'attente, essaie de résoudre ce problème mais sans donner aucune garantie.

Si l'on s'attache aux systèmes temps réel distribués, on est confronté au même problème de surdimensionnement. Classiquement, chercher la garantie absolue de QdS pour des applications sous contraintes temps réel dures revient à prendre en compte du pire cas ; or le système fonctionne en temps normal avec un cas moyen très éloigné du pire cas. Ce phénomène est encore plus accentué avec le déploiement de la méthode d'ordonnancement holistique comme démontré dans [Martin04]. Contrairement à des mécanismes de QdS dans l'Internet qui s'auto-adaptent à l'état du système grâce au mécanisme de contrôle d'admission et des mesures de QdS en-ligne (une sorte de « feedback »), un système temps réel classique conçu selon l'approche temps réel dur souffre de rigidité à cause des hypothèses strictes sur le modèle de tâches/messages. Ce qui peut rendre une solution

ordonnançable vulnérable face aux aléas (de charges, de ressources, de perturbations de l'environnement) Aussi, dans le domaine du contrôle-commande, il apparaît intéressant de s'orienter vers la notion de système adaptatif afin de supporter non seulement la variation de performances du support informatique (tolérance aux fautes), mais aussi l'évolution de l'application qui induit une variation de charges par rapport aux hypothèses du départ sur le modèle d'activation de tâches. En plus, l'utilisation des composants standards, comme par exemple un réseau Ethernet partagé avec d'autres applications à la place d'un bus de terrain, exige aussi l'implémentation des mécanismes de contrôle d'admission et d'ordonnancement de trafics en fonction de la mesure en-ligne de la QdS, c'est à dire avec « feedback ». Fournir des mécanismes de gestion de la QdS appropriés dans un système temps réel adaptatif (incertitude de charges et de ressources) reste encore un problème ouvert [ARTIST03].

Par ailleurs, que ce soit en QdS dans les réseaux ou en ordonnancement dans les systèmes temps réel, les travaux antérieurs négligent le fait que la plupart des applications soient capables, dans une limite à identifier, de tolérer et/ou s'adapter à la variation de performances du système. Des exemples typiques sont la transmission de vidéo et voix qui tolère la perte occasionnelle des paquets, des systèmes de contrôle-commande sur-échantillonnés qui peuvent non seulement tolérer des pertes des échantillons, mais en plus la loi de commande peut aussi compenser des pertes et retards grâce à l'emploi de boucles de contrôle fermées ayant comme entrée supplémentaire la QdS instantanée du système support. Par exemple, des travaux regroupés sous le nom de NCS « Networked Control Systems » visent à étudier la robustesse des lois de commande en fonction de variation de performance de l'architecture informatique support (calculateurs et réseaux) ou à concevoir des lois de commande robuste en prenant en compte la variation de la QdS du système support (en particulier le réseau) [Nilsson98], [Chow01], [Jumel03]. Il est donc plus optimal de concevoir des applications temps réel en prenant compte cette capacité « naturelle » de tolérance du non-respect des échéances. Dans ce cas, le modèle (m,k)-firm [Hamdaoui95] paraît convenable pour spécifier plus précisément les contraintes temps réel. Une contrainte (m,k)-firm est définie sur une tâche récurrente. Elle exige qu'au moins m parmi k invocations consécutives de la tâche doivent être exécutées par le système en respectant leur échéance, avec $m \leq k$ (le cas où $m = k$ est équivalent du cas de temps réel dur, que nous notons aussi par (k,k)-firm). Si l'on considère qu'une application peut accepter une dégradation de service jusqu'à m exécutions avant l'échéance parmi k demandes d'exécutions consécutives quelconques, un système peut alors conçu selon l'approche (m,k)-firm pour offrir des niveaux de QdS variés entre (k,k)-firm (cas normal) et (m,k)-firm (pire cas) avec autant de niveaux intermédiaires correspondant aux différentes valeurs possibles entre k et m. Ce qui résulte en un système avec la dégradation de la QdS contrôlée (Graceful degradation).

La garantie du respect des contraintes temps réel selon le modèle (m,k)-firm dans un système temps réel dynamique et dans un réseau à QdS nécessite des efforts de recherche dans deux directions : 1) pour la prise en compte explicite de cette nouvelle contrainte (m,k)-firm, des algorithmes d'ordonnancement classiques tels que EDF (Earliest Deadline First), FP (Fixed Priority), WFQ (Weighted Fair Queueing) doivent être étendus et des algorithmes nouveaux restent à développer ; 2) bien qu'en moyenne (m,k)-firm permette de diminuer le besoin de ressources par rapport au temps réel dur qui est équivalent à (k,k)-firm, il n'est pas toujours possible de réaliser ce gain lors que l'ordonnancement est non préemptif et une garantie déterministe de (m,k)-firm (appelé aussi par certains chercheurs (m,k)-hard) est exigée. Ceci à cause de la NP-complétude du problème. Deux pistes sont possibles : soit le développement de l'analyse d'ordonnançabilité vis à vis de l'algorithme d'ordonnancement proposé dans des cas particuliers mais représentent un intérêt pratique, soit étendre le modèle (m,k)-firm initial afin de permettre de réaliser ce gain.

L'objectif de ce papier est de donner un aperçu des algorithmes d'ordonnancement pour la garantie déterministe temps réel (m,k)-firm et les appliquer à la gestion de la QdS.

La gestion de la QdS est assurée par trois fonctions fondamentales: *ordonnancement*, *gestion de files d'attente* en cas de saturation et la *régulation de trafic*. Dans ce papier, nous nous intéressons principalement à l'application du modèle (m,k)-firm dans la fonction de l'ordonnancement.

Le reste de ce papier est organisé comme ce qui suit. La section 2 présente un état de l'art sur les travaux autour de (m,k)-firm. La section 3 décrit (m,k)-WFQ qui permet à un serveur WFQ (Weighted Fair Queueing) de prendre en compte plus efficacement des contraintes temporelles des flux multimédias temps réel. La section 4 présente une analyse d'ordonnançabilité de l'algorithme DBP (Distance Based Priority) non préemptif. Enfin, la section 5 conclut le papier et indique les perspectives.

2. Etat de l'art sur (m,k)-firm

2.1. Modèle général: MIQSS

Considérons un modèle d'accès multiple à une ressource partagée que nous allons appeler MIQSS (Multiple Input Queues Single Server) dans la suite de ce document. Nous cherchons à ordonnancer des demandes d'accès au serveur commun, tout en satisfaisant leurs

contraintes temporelles et en optimisant le taux d'utilisation du serveur. Dans notre contexte de systèmes distribués temps réel, ce serveur peut modéliser un processeur pour les demandes d'exécution des invocations de tâches ou un médium de transmission (bande passante) de paquets.

Afin que nos résultats puissent aussi être applicables à la transmission de paquets, seul le cas *non-préemptif* nous intéresse. Notons que ce cas est en général plus difficile à analyser que le cas préemptif.

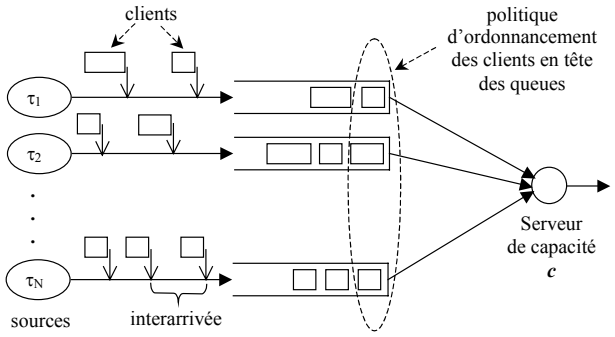


Figure 1. Modèle MIQSS

Une source τ_i est caractérisée par sa fonction de flux d'arrivée F_i. Dans cette étude, cette fonction peut être :

- **Périodique ou sporadique**: décrit par (C_i, T_i) dans le cas d'une date initiale quelconque d'arrivée du premier client et (r_i, C_i, T_i) dans le cas d'une date initiale fixe r_i, où C_i est le temps d'exécution d'un client et T_i la période d'inter-arrivée (ou d'inter-arrivée minimale dans le cas sporadique).
- **Périodique avec gigues** : (C_i, T_i, J_i) ou (r_i, C_i, T_i, J_i). Où J_i représente le déphasage maximum de l'instant d'une arrivée de client par rapport à la période.
- **(σ_i, ρ_i)-borné** : une courbe linéaire caractérisée par une taille de rafale σ_i et un débit moyen ρ_i qui majore la vraie fonction cumulative d'arrivée du travail [LeBoudec02], [Chang00]. La quantité du travail apportée par un client est définie par W_i avec notamment $C_i = W_i / c$ où c représente le débit du serveur.

Dans la suite, les contraintes temps réel sont toujours données par (D_i, m_i, k_i) où D_i est l'échéance relative à l'instant d'arrivée d'un client et (m_i, k_i) sont les deux paramètres de la contrainte (m_i, k_i)-firm.

2.2. Expression de contraintes (m,k)-firm et WHRT

Une source sous contrainte temps réel (m, k)-firm peut se trouver dans l'un des deux états : normal et échec transitoire (*dynamic failure*) [Hamdaoui95]. La connaissance de son état à l'instant t dépend de l'historique du traitement des k derniers clients générés par la source. Si l'on associe '*1*' à un client respectant son échéance et '*0*' à un client ratant son échéance, cet historique est alors entièrement décrit par une suite de k bits appelée une *k-séquence*. La Figure 2 donne un exemple de (2,3)-firm avec par convention le déplacement vers la gauche des bits.

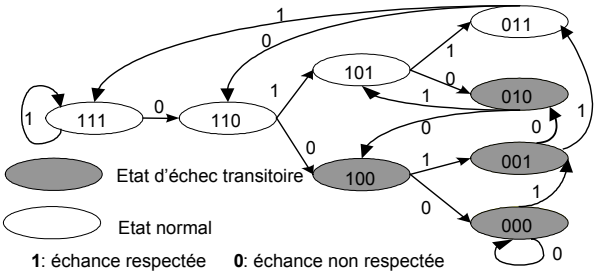


Figure 2. Diagramme d'état-transition d'une source avec (2,3)-firm

Dans un système qui peut être modélisé par MIQSS, on peut définir l'état du système à un instant t à partir des états des sources du même instant. Un système est dit en état d'échec transitoire si au moins une de ses sources est en échec transitoire (une sorte de ET logique entre les états de l'ensemble de sources).

Une source peut exprimer sa contrainte (m,k)-firm en spécifiant simplement la valeur des deux paramètres : m et k.

Afin de faciliter l'expression des contraintes du type (m,k)-firm mais avec plus de précision sur la répartition des m parmi les k clients consécutifs, [Bernat97] et [Bernat01] ont enrichi ce modèle (m,k)-firm en proposant trois autres formes qui correspondent à la complémentarité et la consécutivité:

- $\left(\overline{m}, \overline{k}\right)$-firm : au plus m clients avec *échéance non respectée* dans une fenêtre quelconque de k arrivées consécutives
- $\langle m, k \rangle$-firm : au moins m clients *consécutifs* avec échéance respectée dans une fenêtre quelconque de k arrivées consécutives
- $\langle \overline{m}, \overline{k} \rangle$-firm: au plus m clients *consécutifs* avec *échéance non respectée* dans une fenêtre quelconque de k arrivées consécutives

La notion de (m,k)-firm est alors généralisée et une source sous ces formes de contraintes est dite sous contrainte WHRT (Weakly-Hard Real-Time) [Bernat01]. Néanmoins il convient de remarquer que certaines de ces formes peuvent toujours être exprimées sous forme de (m,k)-firm :

- $\left(\overline{m}, \overline{k}\right)$-firm : équivalente à (k-m,k)-firm

- $\langle m, k \rangle$-firm : pas d'équivalence dans (m,k)-firm
- $\langle \overline{m}, \overline{k} \rangle = \langle \overline{m} \rangle$: en fait il est facile de constater qu'avec $\langle \overline{m}, \overline{k} \rangle$, on ne peut jamais avoir plus de *m* clients *consécutifs* avec *échéance non respectée* quelque soit la taille de *k* pourvu que $m < k$. De plus une source respectant (m,k)-firm inclut le cas particulier de $\langle \overline{k-m} \rangle$.

Notons que la k-séquence réalisée par un algorithme d'ordonnancement n'est pas nécessairement répétitive. On parle alors de *k-séquence dynamique*.

Un cas particulier d'expression de contrainte (m,k)-firm est la spécification d'une *k-séquence fixe* appelée un κ-pattern (ou (m,k)-pattern [Quan00]). Cette technique s'inspire du modèle de calcul imprécis [Chung90] où une tâche est composée d'une partie critique (mandatory) et d'une partie optionnelle. Le κ-pattern d'une source ayant une contrainte temporelle (m,k)-firm est défini par la succession de *k* éléments de l'alphabet {0, 1} où '0' indique une demande de traitement optionnelle et '1' une demande critique avec $\sum_{i=1}^{k} \pi_i = m$ où π_i est le $i^{ème}$ élément du κ-pattern pour $1 \le i \le k$.

En répétant continuellement le κ-pattern, on classe les demandes de traitement des clients d'un flux (ou une source) en deux catégories : optionnelle et critique. Il est facile de prouver qu'il suffit de traiter avec succès toutes les demandes critiques (les *m* « 1 ») pour satisfaire la contrainte (m,k)-firm (voir [Ramanathan99], Théorème 1). Notons que la réciproque n'est pas vraie car une garantie (m,k)-firm n'a pas objectif de produire une k-séquence fixe. Les demandes optionnelles peuvent être traitées quand le serveur n'est pas occupé ou rejetées si leur échéances ne peuvent pas être respectées par le serveur.

De ce fait, le $n^{ième}$ client (ou demande de traitement) d'un flux ayant la contrainte temporelle (m,k)-firm est considéré comme étant un client critique si et seulement s'il satisfait la relation suivante :

$$\pi_{(n\%k)} = 1 \qquad (1)$$

avec *n%k* le reste de la division de *n* modulo *k*.

L'utilisation d'un κ-pattern fixe a l'avantage de ramener le problème de l'analyse d'ordonnançabilité du système (m,k)-firm à celui de l'analyse d'ordonnançabilité classique. Par exemple quand tous les clients critiques sont ordonnancés sous la politique FP (fiexed priority) et les clients optionnels ont la priorité la moins élevée, l'analyse d'ordonnançabilité est donnée dans [Ramanathan99]. L'application de cette classification peut être utile dans le domaine du multimédia. En effet, ce concept peut être appliqué à un flux de paquets vidéos pour sélectionner les paquets critiques dans un GOP (Group of Pictures) en utilisant le standard de compression MPEG [Furht99]. Par exemple, un flux compressé utilisant la structure du GOP suivante [*IBBPBBPBB*], où les paquets I (Intra images) et P (Predicted images) sont plus importants que les paquets B (Bi-directional predicted/interpolated images), peut être considéré comme étant un flux ayant des contraintes temporelles de type (6,9)-firm et spécifié par le κ-pattern suivant $\{\pi_{i(1 \le i \le k)}\} = \{110110110\}$. Ce κ-pattern signifie qu'une partie des paquets de type B est déclarée comme optionnelle par la source de ce flux. Par exemple, le $226^{ème}$ paquet est considéré comme étant critique car $\pi_{(226\%9)} = \pi_1 = 1$ et le $228^{ème}$ paquet est considéré comme étant optionnel car $\pi_{(228\%9)} = \pi_3 = 0$.

Une fois la contrainte WHRT spécifiée, on peut alors passer à l'étape de recherche d'algorithmes d'ordonnancement pour que la contrainte soit respectée (de façon déterministe ou probabiliste).

2.3. Algorithmes d'ordonnancement pour (m,k)-firm

Il existe aujourd'hui principalement deux familles d'algorithmes qui prennent en compte (m,k)-firm: dynamique (par exemple DBP : Distance Based Priority) et statique (par exemple EFP : Enhanced Fixed Priority). Par algorithme dynamique nous voulons dire que la priorité affectée à chaque client est ajustée automatiquement en fonction de l'état du système (en particulier de la *k-séquence* des sources) à l'instant *t*. Tandis qu'une affectation statique de priorité est basée sur un paramètre fixe (taux *m/k* par exemple).

Un algorithme dynamique a l'avantage de permettre au système de s'adapter aux changements de situation (variation de flux, de capacité du serveur, …). Il convient à la gestion en-ligne de la QdS. Le problème est qu'il ne donne souvent qu'une garantie statistique (best-effort) de *m* sur *k*. C'est le cas de DBP et de la première version de DWCS (Dynamic Window Constrained Scheduling) [West99]. Une version améliorée de DWCS [West04] permet de donner une garantie déterministe de *m* sur *k* sous des conditions particulières (même C_i pour toutes les sources). *A contrario*, un algorithme statique permet une vérification hors-ligne du système et garantit de façon déterministe le respect de *m* sur *k* échéances dans le cas où le système ne violerait pas les hypothèses du pire cas.

Dans ce qui suit nous expliquons le principe de DBP, DWCS et EFP.

2.3.1. DBP (Distance Based Priority)

DBP [Hamdoui95] est la façon la plus directe pour la prise en compte de la contrainte (m,k)-firm. Pour une k-séquence donnée, DBP définit à chaque début du service d'un client la distance d'aller à un état d'échec transitoire comme le nombre consécutif de bits 0 à ajouter pour atteindre cet état. La priorité que DBP donne au client en tête de la queue correspondante à la k-séquence est égale à cette distance. Si la source se trouve déjà en état d'échec transitoire (i.e., moins de *m* 1 dans la k-séquence), la plus haute priorité 0 est affectée. Par exemple, pour une source sous contrainte (3,5)-firm, le client en tête de la queue est de priorité 2 si les 5 clients précédents forment une k-séquence (11011), il est de priorité 3 si les 5 clients précédents forment une k-séquence (10111).

Formellement, selon [Hamdaoui95] la priorité est évaluée comme suit. Nous notons par $s_j = \left(\delta^j_{i-k_j+1}, \cdots, \delta^j_{i-1}, \delta^j_i\right)$ la k-séquence de la source τ_j, par $l_j(n,s)$ la position (en comptant à partir de droite) de la n^{ième} échéance respectée (ou 1) dans s_j, la priorité du (i+1)^{ème} client de τ_j est donnée par :

$$P\_DBP^j_{i+1} = k_j - l_j\left(m_j, s_j\right) + 1 \qquad (2)$$

Notons que lorsqu'il y a moins de *n* 1 dans *s*, alors $l_j(n,s) = k_j + 1$, afin que la plus haute priorité (= 0) soit affectée.

La Figure 3 schématise comment DBP est utilisé pour l'affectation de priorité. Cette politique d'affectation dynamique de priorité peut être facilement et efficacement implémentée en matériel car l'historique de chaque source peut être stocké dans un registre de k_j bits.

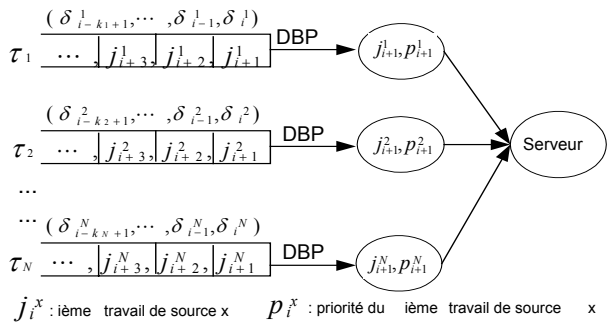


Figure 3. DBP pour l'affectation de priorité des clients en tête des queues

Le serveur choisit les clients présents en tête des queues selon leur priorité. Dans le cas d'égalité de priorité parmi les clients à choisir, EDF (Earliest Deadline First) est utilisée par défaut. Nous notons par DBP-EDF ce système.

2.3.2. DWCS (Dynamic Window Constrained Scheduling)

L'algorithme DWCS a été conçu dans [West99] pour maximiser l'utilisation de la bande passante du réseau en cas de surcharge pour des flux temps-réel tolérant aux pertes. Il se charge de garantir la contrainte de type $(\overline{2x}, \overline{x+y}) - firm$, c'est-à-dire, pas plus que $2x$ dépassements d'échéances dans n'importe quelle fenêtre de $x+y$ paquets consécutifs tout en ayant la capacité de partager la bande passante entre les paquets des flux en compétition en proportion de leurs échéances et tolérances aux pertes, avec x représente le nombre de paquets qui pourraient être perdus ou transmis en retard pour chaque fenêtre fixe de taille y paquets consécutifs. DWCS est développé pour être employé comme une alternative à EDF dans des conditions de surcharge, étant donné que les performances de EDF se dégradent sérieusement pour une charge supérieure à un.

Cet algorithme nécessite deux attributes par flux pour assurer l'ordonnancement des paquets :

- *L'échéance* D_i : elle est définie comme étant le temps maximum entre le service de deux paquets consécutifs au sein d'un même flux. Dans le cas d'un flux périodique, l'échéance D_i d'un flux τ_i est égale à sa période T_i.
- *La contrainte de fenêtre fixe:* elle est aussi appelée *facteur de tolérance aux pertes*. Elle est spécifiée par la valeur $W_i = x_i / y_i$ où x_i représente le nombre maximum de paquets perdus (ou transmis en retard) pour chaque fenêtre fixe de taille y_i paquets consécutifs.

Bien que DWCS s'intéresse à une fenêtre fixe, cette contrainte peut inclure le cas de fenêtre glissante du modèle (m,k)-firm. Dans [West04], il a été montré que cette contrainte (*x,y*) permet, au pire (quand les *x* paquets perdus se trouvent à la fin d'une fenêtre de taille *y* et les *x* autres paquets se perdent au début de la fenêtre suivante), de garantir le respect de $(\overline{2x}, \overline{x+y}) - firm$. Comme DBP, DWCS maintient l'information d'état par flux mais l'utilisation de cette information diffère significativement de DBP. En effet, DBP affecte la priorité relative à un flux en se basant sur l'historique des *k* derniers clients, alors que DWCS utilise la notion de la fenêtre fixe dans laquelle *x* et *y* changent de valeurs au cours du temps selon un algorithme que nous expliquons par la suite.

DWCS choisit les paquets à servir en fonction de leurs échéances ainsi que leurs facteurs de tolérance aux pertes. Dans [West99], l'affectation de priorité selon la

première version de DWCS (DWCS[1]) se résume en six règles et est présentée dans le tableau suivant.

1	Choisir le paquet avec la plus petite contrainte de fenêtre (*plus petit facteur de tolérance aux pertes*) $= \min_{i=1..N}\left(W_i = x_i / y_i\right)$ avec $y_i \neq 0$
2	S'il existe $1 \le i, j \le N \ / W_i = W_j \neq 0$, alors servir avec EDF $= \min_{n=1..N}\left(D_n\right)$
3	S'il existe $1 \le i, j \le N \ / W_i = W_j \neq 0$ et $D_i = D_j$, alors servir le paquet ayant le plus petit numérateur de la contrainte de fenêtre $= \min_{i=1..N}\left(x_i\right)$
4	Si $W_i = W_j = 0$ et $y_i = y_j = 0$, alors servir avec EDF $= \min_{n=1..N}\left(D_n\right)$
5	Si $W_i = 0$, alors servir le paquet ayant le plus grand dénominateur de la contrainte de fenêtre $= \max_{n=1..N}\left(y_n\right)$
6	Tous les autres cas sont traités par FIFO

Nous observons que si deux paquets ont les mêmes valeurs de facteurs de tolérance aux pertes et les mêmes valeurs d'échéances, alors les paquets sont servis selon l'ordre croissant des x_i où x_i / y_i représente la valeur *courante* du facteur de tolérance aux pertes pour tous les paquets du i[ème] flux. Ainsi, la priorité est affectée au paquet du flux ayant la contrainte de perte la plus étroite, afin d'éviter des pertes consécutives de paquets. Si les facteurs de tolérance ainsi que les dénominateurs y_i des deux paquets sont nuls, alors les paquets sont servis dans l'ordre croissant de leurs échéances ; Sinon, si les dénominateurs y_i sont non nuls, alors le paquet ayant la plus grande valeur du dénominateur de la contrainte de fenêtre sera affecté la plus haute priorité.

Chaque fois qu'un paquet du flux *i* est transmis, la contrainte de fenêtre du i[ème] flux est ajustée. De même, les contraintes de fenêtre des autres flux sont ajustées dans le cas où il existe des paquets de ces flux qui ont dépassé leurs échéances.

Pour les flux tolérant les pertes de paquets, les paquets ayant raté leurs échéances sont tout simplement rejetés. Pour les flux ne tolérant pas de pertes de paquets, les échéances servent à réduire le délai d'attente dans les files avant leur transmission. La valeur du facteur de tolérance sert dans ce cas à éviter un retard excessif des paquets de tel flux.

Les contraintes x et y sont ajustées au cours du temps en fonction des échéances si elles sont ratées ou non. Considérons un flux *i* ayant la contrainte de fenêtre originale $W_i = x_i / y_i$ à l'instant initial. Notons par $W_i' = x_i' / y_i'$ la contrainte de fenêtre courante. Si le paquet du flux *i* est transmis avant le dépassement de son échéance, les contraintes x_i' et y_i' sont ajustées de la façon suivante :

$$\begin{cases} \text{si } \left(y_i' > x_i'\right) \text{ alors } y_i' = y_i' - 1 \\ \text{si } \left(x_i' = y_i' = 0\right) \text{ alors } x_i' = x_i \ ; y_i' = y_i \end{cases}$$

Cependant, pour tous les paquets des autres flux en attente, si un paquet du flux $j / j \neq i$ rate son échéance, alors les contraintes sont ajustées selon la règle suivante:

$$\begin{cases} \text{Si } \left(x_j' > 0\right) \text{Alors} \\ \quad \left| \begin{array}{l} x_j' = x_j' - 1;\ y_j' = y_j' - 1; \\ \text{Si } \left(x_j' = y_j' = 0\right) \text{ Alors } x_j' = x_j;\ y_j' = y_j \end{array} \right. \\ \text{Sinon Si } \left(x_j' = 0\right) \text{ Alors} \\ \quad \left| \begin{array}{l} \text{Si}\left(x_j > 0\right) \text{ Alors } y_j' = y_j' + \left\lceil \dfrac{y_j + x_j}{x_j} \right\rceil \\ \text{Si}\left(x_j = 0\right) \text{ Alors } y_j' = y_j' + y_j \end{array} \right. \end{cases}$$

Donc à chaque fois qu'une échéance du flux *j* est ratée, le facteur de tolérance aux pertes de ce flux est ajusté de façon à lui donner plus d'importance dans le prochain tour de sélection de paquet. Cette approche évite le problème de famine en affectant des priorités plus élevées aux flux qui sont susceptibles de violer leurs contraintes de fenêtre initiales. Inversement, un paquet du flux *i* est servi avec respect de son échéance, conduit à la diminution du facteur de tolérance des autres flux réduisant ainsi sa priorité aux prochains tours.

Récemment, West et al. proposent dans [West04] la deuxième version de DWCS (DWCS[2]). La différence principale avec la première version est que les deux premières lignes du tableau sont inversées. Dans la deuxième version de DWCS, la première règle d'affectation de priorité est identique à EDF, i.e. le paquet ayant la plus petite échéance est le plus prioritaire. La deuxième règle dans DWCS[2] fait recours à une comparaison des contraintes de fenêtre lorsque les échéances sont égales. West et al. expliquent que le changement de l'ordre des règles est dû à l'optimalité de EDF dans des conditions de charge normale pour respecter les échéances et par conséquent les contraintes de fenêtre. Cependant, l'algorithme DWCS[1] reste toujours plus performant que EDF dans des conditions de surcharge où il est impossible de respecter toutes les échéances.

Dans [West04], les auteurs étudient les caractéristiques temporelles de DWCS[2] et montre analytiquement que, dans le cas où il existe un ordonnancement faisable pour un ensemble de flux périodiques, les délais des flux en service sont toujours

bornés même en situation de surcharge. En effet, il a été montré que le délai garanti à chaque flux est indépendant des autres flux en service même en situation de surcharge. De plus, les résultats de simulation montrent que DWCS et DBP on des performances similaires en termes de nombre d'échéances ratées et de violation de la contrainte de fenêtre. Enfin, une implémentation sur Linux de DWCS est téléchargeable à partir du site de l'auteur.

2.3.3. EFP (Enhanced Fixed Priority)

EFP est proposé dans [Hamdaoui97], [Ramanathan99]. Pour prendre en compte la contrainte (m,k)-firm, il suffit que chaque source définisse un κ-pattern et marque parmi ses k clients consécutifs m clients *critiques* et k-m clients *optionnels*. En faisant ainsi le serveur pourra rejeter un client optionnel en cas de surcharge (c'est à dire au cas où son échéance ne peut plus être respectée par le serveur). Tous les clients critiques peuvent être ordonnancés par un algorithme à priorité fixe tel que RM (Rate Monotonic). Les clients optionnels sont servis avec la priorité la plus basse selon la politique FIFO. Le problème revient donc à définir un κ-pattern. Pour commencer le marquage, le premier client de chaque source est marqué critique par défaut. Pour une source τ_i, le marquage des clients critiques et optionnels selon sa contrainte (m_i,k_i)-firm est alors entièrement donné par l'équation suivante.

Le $n^{\text{ième}}$ client ($n = 0, 1, \ldots$) est marqué critique si n vérifie : $n = \left\lfloor \left\lceil \frac{n \times m}{k} \right\rceil \times \frac{k}{m} \right\rfloor$

Ce qui donne comme κ-pattern suivant (pour $i=1, 2, \ldots, k$) :

$$\pi_i = \begin{cases} 1 & \text{si } \; i = \left\lfloor \left\lceil \frac{i \times m}{k} \right\rceil \times \frac{k}{m} \right\rfloor \\ 0 & \text{sinon} \end{cases} \qquad (3)$$

Le marquage ne dépend que du rapport m_i/k_i. Une condition suffisante est donnée dans [Ramanathan99] pour la garantie déterministe de contrainte (m_i,k_i)-firm.

Cet algorithme souffre néanmoins trois problèmes:

- Le premier client de chaque source est marqué critique par défaut. Ce qui force artificiellement le système de se retrouver dans un « pire cas ».
- L'équation 3 distribue régulièrement les m clients critiques parmi les k arrivées consécutives. Ce qui peut ne pas être optimal dans certaines situations.
- La technique de marquage ne dépend que du rapport m_i/k_i, mais pas de C_i et T_i. Deux sources ayant des C_i et T_i très différents mais avec la même contrainte (m,k)-firm relèveront du même κ-pattern et donc se verront leur clients critiques distribués de la même façon. Le fait de ne pouvoir les distinguer peut conduire à une situation non optimale.

Partant de l'idée qu'une partition judicieuse et globale des clients critiques de toutes les sources devrait donner une meilleure ordonnançabilité, [Quan00] a amélioré l'algorithme présenté dans [Hamdaoui97, Ramanathan99]. Il a d'abord prouvé que trouver une partition optimale est NP-difficile. Puis, il donne une heuristique pour optimiser la répartition de m_i clients critiques parmi k_i clients consécutifs en prenant en compte les relations entre les sources.

3. (m,k)-WFQ pour une meilleure gestion de la QdS temps réel

L'ordonnanceur WFQ (Weighted Fair Queueing) [Parekh93] est déployé dans les commutateurs et routeurs du réseau Internet pour fournir de la QdS grâce à ses propriétés de garantie de bande passante et de délai borné pour des flux (σ,ρ)-bornés. L'algorithme (m,k)-WFQ consiste à intégrer les contraintes temporelles (m,k)-firm au processus d'ordonnancement de WFQ. Nous faisons d'abord un rappel du principe de WFQ afin d'expliquer ensuite l'apport de (m,k)-WFQ.

WFQ garantit à chaque flux servi la proportion de la bande passante réservée selon son coefficient de partage Φ_i. Chaque paquet de messages est estampillé par un tag appelé temps virtuel de départ. Le serveur sélectionne toujours le paquet dont le temps virtuel de départ est le premier à partir de l'instant de sélection. Dans WFQ le temps virtuel de départ est défini par :

$$F_i^k = \max\left\{F_i^{k-1}, V(t)\right\} + \frac{L_i^k}{\Phi_i} \qquad (4)$$

avec

- F_i^k : temps virtuel de départ du $k^{ième}$ paquet du $i^{ème}$ flux,
- V(t) : le temps virtuel quand le $k^{ième}$ paquet arrive,
- Φ_i : le coefficient de partage du $i^{ème}$ flux,
- L_i^k : la taille du $k^{ième}$ paquet du $i^{ème}$ flux,
- $\max\left\{F_i^{k-1}, V(t)\right\}$: le temps virtuel du début de service du $k^{ième}$ paquet.

Avec WFQ, il est montré dans [LeBoudec02] que pour un flux τ_i de type (σ_i,ρ_i)-borné et ayant un débit moyen réservé $g_i \geq \rho_i$, le délai garanti par WFQ à ce flux est borné par :

$$D_{i,\max} = \frac{\sigma_i}{g_i} + \frac{L_{\max}}{c} \qquad (5)$$

où L_{max} est la taille maximale du paquet parmi tous les paquets dans tous les flux et c la capacité de traitement du serveur.

Nous rappelons qu'un flux est dit (σ,ρ)-borné si sa fonction cumulative d'arrivée $R(t)$ vérifie la relation $\forall 0 \le s \le t,\ R(t) - R(s) \le \sigma + \rho(t-s)$ avec σ la taille maximale de rafale et ρ le débit moyen à long terme.

La borne fournie par WFQ sur le temps de réponse d'une source de flux est étroitement liée au coefficient de partage de la bande passante ρ_i et à la taille de la rafale σ_i. Pour avoir un délai d'attente court, un flux doit réserver une large bande passante. Pour un flux de faible débit moyen et ayant une grande rafale ceci peut conduire à une mauvaise utilisation de la bande passante. Ce problème peut être résolu avec la politique WFQ priorisé proposé dans [Wang02] mais la notion de (m,k)-firm n'est pas prise en compte.

Nous avons proposé dans [Koubâa04a], [Koubâa04b] une approche appelée (m,k)-WFQ. Pour que l'ordonnanceur WFQ puisse prendre en compte les contraintes temporelle (m,k)-firm, nous exprimons la contrainte par un κ-pattern, donc la source marque m paquets critiques parmi tous les k paquets consécutifs et les autres étant optionnels. L'ordonnanceur (m,k)-WFQ estampille ensuite le paquet par son temps virtuel de départ décrit par l'équation 4. L'algorithme est décrit dans la Figure 4. Le processus de service est activé quand au moins un paquet existe dans la file d'attente du système. Le serveur sélectionne le paquet ayant le plus petit temps virtuel de départ parmi tous les paquets critiques présents en tête de files. Si aucun paquet critique existe, le choix sera fait parmi les paquets optionnels. Puis, si le paquet sélectionné est critique, il est exécuté (ou transmis) directement par le serveur, tandis que si le paquet est optionnel, l'ordonnanceur vérifie avant son exécution si ce paquet pouvait éventuellement satisfaire son échéance. Si l'échéance souhaitée ne peut être garantie après l'exécution, le serveur rejette le paquet et refait une nouvelle sélection, sinon, il l'envoie.

L'avantage de l'algorithme proposé est qu'il permet de garantir une bande passante à un flux tout en intégrant les propriétés temporelles dans le processus d'ordonnancement ce qui revient à gérer les flux plus efficacement. En effet, le rejet des paquets optionnels qui ne satisfont pas leurs échéances permet au serveur de donner la main plus rapidement aux paquets critiques en attente. Cette perte ne dégrade pas les performances des flux servis tant que leurs contraintes (m_i,k_i)-firm sont satisfaites. Ainsi, (m,k)-WFQ diminue forcément les bornes sur les temps de réponse des flux temps réel par rapport à WFQ standard. Dans ce qui suit nous montrons quantitativement cette amélioration par simulation d'un exemple.

```
Entrées
Flux τ_i = {(Période ou Débit), Echéance Désirée, (m_i,k_i), (Gigue ou Rafale), Taille de Paquet)

Affectation de priorité
A l'arrivée du a^ième du flux [i] {
  si (π(a%k_i)=1) alors {
      Marquer le paquet comme critique;
      }
  sinon {
      Marquer le paquet comme optionnel;
      }
  Calculer le temps virtuel de départ F_i^k ;
  Estampiller le paquet avec F_i^k ;
  Mettre le paquet dans sa file d'attent;

Discipline de Service
Serveur = libre;

   si (serveur!= occupé) {
     Choisir le paquet dont F_i^k plus petit
      si (paquet est critique) {
         Envoyer le paquet;
         Serveur = occupé;
         }

            si (l'échéance serait ratée){
                 Rejet du paquet;
                 Serveur = libre;
                 }

                 Envoyer le paquet;
                 Serveur = occupé;
                 }
         }
      }
   si (serveur== occupé) {
    attendre jusqu'à tx totale du paquet;
    Serveur= libre;
  }
```

Figure 4. Algorithme (m,k)-WFQ

Considérons un réseau constitué de trois sources de trafic. Ces trois sources partagent un lien de 10 Mbit/s selon leurs coefficients de réservation. Dans cette simulation, on considère une taille fixe à tous les paquets des trois flux de 8 *Kbits*. Le Tableau 1 récapitule les paramètres de simulation pour chacun des flux.

Le marquage de paquets en critiques et optionnels est spécifié par un κ-pattern fixe pour chaque source.

La première source génère un flux de voix selon le modèle de trafic ON/OFF. Les périodes d'activité ON et de silence OFF sont exponentiellement distribuées avec les moyennes $1/\mu_{ON} = 500ms$ et $1/\mu_{OFF} = 755ms$ avec une période de génération de paquets dans la période d'activité de *50 ms*. Donc, le débit moyen du flux est de 64 Kb/s. Les contraintes temporelles sont de type (4,5) et l'échéance souhaitée d'un paquet est fixée à 10 ms. Le κ-pattern fixe le profil de la séquence comme : `11011 11011 11011 …`

	(m,k)	Débit	Trafic	κ-pattern	Echéance
Voix	(4,5)	64 kb/s	ON/OFF (500/755/50)ms	*11011*	10 ms
Vidéo	(3,5)	2Mb/s	Périodique avec gigue ~2Mb/s	*10110*	4 ms
FTP	(0,1)	7,936 Mb/s	Périodique avec gigue ~7.936 Mb/s	*0*	Infinie

Tableau 1. Configuration simulée

La deuxième source est une source CBR (Constant Bit Rate) périodique avec gigue (95% de T_i-C_i) qui génère un flux vidéo de 2 Mbit/s. L'échéance des paquets est fixée à 4 ms avec une garantie de type (3,5). Le κ-pattern fixe le profil de la séquence comme : `10110 10110 10110 …`

La troisième source est un agrégat de flux FTP, que nous supposons périodique avec gigue (95% de T_i-C_i) et qui consomme le reste de la bande passante ayant donc un débit de 7,936 Mb/s. Un flux FTP est vulnérable à la perte de paquets et ce trafic fonctionne en mode Best-Effort. Donc, il ne possède pas de propriétés temporelles strictes comme dans le cas des deux sources temps-réel : Voix et Vidéo. Par conséquent, nous fixons une garantie de type (0,1) pour le flux FTP et une échéance infinie afin d'éviter tout rejet de paquets FTP optionnels.

Le tableau 2 montre les bornes mesurées sur le temps de réponse des paquets pour chacun des flux et ce pour le serveur (m,k)-WFQ, le serveur WFQ, le serveur (m,k)-FIFO et le serveur FIFO.

	(m,k)-WFQ	WFQ	(m,k)-FIFO	FIFO
Voix	9,769 (*taux de rejet = 6,8%*)	2428,031	20,529	48,031
Vidéo	3,999 (*taux de rejet = 5,5%*)	55,391	21,086	49,031
FTP	9,696	36,562	21,442	49,083

Tableau 2. Bornes sur les temps de réponse (ms)

Les cas du serveur FIFO et (m,k)-FIFO sont simulés pour que l'on puisse les comparer avec le cas du serveur (m,k)-WFQ. Un serveur (m,k)-FIFO est simplement un serveur FIFO avec le rejet des paquets optionnels ayant leur échéances ratées.

Comme prévu, (m,k)-WFQ fournit une garantie plus étroite sur le délai pour les flux temps-réel. Dans ce scénario, on peut remarquer que le délai maximal garanti par WFQ au trafic de la voix est assez grand. Ce résultat découle de deux facteurs majeurs (*cf. équation 5*) : le faible taux de bande passante réservée (64 Kbit/s) et la taille importante de la rafale. L'algorithme (m,k)-WFQ permet de réduire considérablement les bornes sur les temps de réponse en sacrifiant quelques paquets optionnels selon les contraintes temporelles (m,k)-firm de chaque flux. Le rejet des paquets optionnels ne satisfaisant pas leurs échéances améliore nettement le délai des paquets critiques. En comparant (m,k)-WFQ avec la politique (m,k)-FIFO, on peut aussi constater que (m,k)-WFQ conserve la bonne propriété de WFQ en terme de distinction des flux (garantie par flux).

Pour fournir la garantie déterministe de (m,k)-firm dans (m,k)-WFQ, nous donnons la borne sur le temps de réponse de (m,k)-WFQ. L'évaluation de cette borne n'est pas triviale essentiellement à cause de la difficulté de déterminer la part de paquets optionnels que le serveur a effectivement servi. La Figure 5 montre le modèle en « network calculus » qui a permis le calcul de cette borne.

Le calcul de la borne sur le temps de réponse utilise le formalisme du Network Calculus [LeBoudec02]. Dans [Koubâa04a] nous avons intégré les contraintes (m,k)-firm dans le formalisme du Network Calculus en introduisant la notion du (m,k)-

filtre qui permet de filtrer tous les paquets optionnels et fournir en sortie seulement les paquets critiques. La Figure 5 montre la technique pour modéliser le *flux effectif* qui devra être servi par un serveur, garantissant un débit fixé tel que celui de WFQ. Le flux effectif contient tous les paquets critiques et le nombre maximum de paquets optionnels qui pourront être servis par l'ordonnanceur. Les paquets optionnels servis sont ceux qui ne ratent pas leurs échéances. Ce flux effectif est utilisé pour le calcul de la borne sur le délai garanti par (m,k)-WFQ.

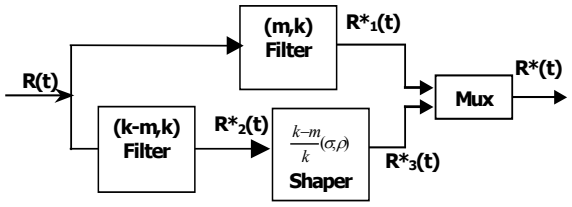


Figure 5. Modèle de Network calculus

Le délai maximal garanti pour une source (σ,ρ)-borné respectant une contrainte temporelle (m,k)-firm avec un taux de partage de bande passante $g \geq \rho$ et servi par un ordonnanceur (m,k)-WFQ est :

$$D_{\max}^{*} = \lambda_{m,k} \cdot \frac{\sigma}{g} + \lambda_{k-m,k} \cdot \frac{e}{g} + \frac{L_{\max}}{c} \qquad (6)$$

Avec $e \leq \sigma$ la taille maximale de rafale des paquets optionnels qui pourraient être transmis par l'ordonnanceur. $\lambda_{m,k}$ désigne le taux de bits critiques du flux et $\lambda_{k-m,k}$ le taux de bits optionnels du flux. Dans le cas où la taille du paquet est constante $\lambda_{m,k} = \frac{m}{k}$. Si aucun paquet optionnel n'est servi, $D_{\min}^{*} = \lambda_{m,k} \cdot \frac{\sigma}{g} + \frac{L_{\max}}{c}$ est la plus petite borne sur le délai. Pour garantir un délai entre D_{min}* et D_{max}*, on peut ajuster l'échéance maximale D_{op} qui détermine $e = gD_{op}$.

L'algorithme (m,k)-WFQ peut être étendu et intégré dans Intserv et le réseau ATM. L'idée de base est que chaque source voulant profiter d'une garantie avec dégradation adaptée doit marquer ses paquets en tant que optionnel ou critique selon sa contrainte (m,k)-firm et son κ-pattern associé. L'ordonnanceur WFQ qui garantit le débit dans le cadre du service garanti, doit tenir compte de cette classification. Les paquets optionnels dont l'échéance ne peut être respectée sont rejetés. (m,k)-WFQ permet alors de garantir des bornes sur le délai plus précises et d'une manière plus flexible. Pour une source ayant un trafic défini par le TSPEC (M,p,b,r) de Intserv et d'ATM avec *M* la taille maximale d'un paquet, *p* le débit crête, *b* la taille maximale de la rafale autorisée et *r* le débit moyen à long terme associé à la contrainte (m,k)-firm et autorisant un délai maximal pour les paquets optionnels égal à D_{op}, le délai maximal D_{max} a été obtenu dans [Koubâa04b] de façon similaire à l'obtention de l'équation 6.

4. Garantie déterministe et condition suffisante de DBP

On vient de voir que beaucoup d'algorithmes d'ordonnancement ont été proposés pour fournir une garantie en moyenne (best-effort) et déterministe du temps réel (m,k)-firm. S'il est vrai que par rapport à la garantie déterministe du temps réel dur, le fait de ne plus viser que garantir en moyenne *m* échéances parmi les *k* instances consécutives d'une tâche résulte en moins de demande de ressources en moyenne, il n'est pas évident que cet avantage est toujours conservé lorsqu'on cherche une garantie déterministe de (m,k)-firm. Cette question est fondamentale pour savoir si un algorithme d'ordonnancement pour (m,k)-firm peut apporter des avantages par rapport à un algorithme connu (EDF, FP, …) pour le temps réel dur avec garantie déterministe. Le point clé pour répondre à cette question est la recherche de conditions suffisantes d'ordonnançabilité. Un ensemble de sources $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ (dans le modèle MIQSS) ordonnançable respecte alors la contrainte (m,k)-firm de façon déterministe car l'analyse d'ordonnançabilité est réalisée dans le pire cas.

Le cas de (m,k)-WFQ donne relativement simplement cette garantie déterministe grâce à WFQ qui transforme en fait un serveur partagé en *N* serveurs dédiés à *N* sources, avec comme facteur d'interférence la longueur maximale d'un paquet L_{max}. L'obtention d'une condition suffisante dans un modèle MIQSS avec non préemption est en général un problème difficile.

Dans ce paragraphe, nous donnons d'abord un état de l'art sur ce problème de recherche de conditions suffisantes pour l'ordonnancement non préemptif, puis une condition suffisante pour la garantie déterministe du temps réel (m,k)-firm avec l'ordonnancement NP-DBP-EDF (Non Preemptive - Distance Based Priority - Earliest Deadline First) [Li03], [Li04].

4.1. Etat de l'art sur les conditions suffisantes

Nous commençons par nous intéresser à la condition suffisante pour la garantie déterministe (k,k)-firm (i.e. temps réel dur). Pour un ensemble de sources périodiques ou sporadiques $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$

avec $\tau_i = \{T_i, C_i, D_i\}$ et des dates initiales quelconques, [Jeffay91] a donné un ensemble de conditions suffisantes et nécessaires d'ordonnançabilité sous EDF non préemptif (noté par NP-EDF : Non-Preemptive EDF). Dans la suite de ce paragraphe nous supposons que le temps est discrétisé et indexé par les entiers. Nous supposons également que l'échéance est égale à la période (ou à l'intervalle d'interarrivée minimal s'il s'agit du cas sporadique).

Théorème de [Jeffay91] :

Considérons un ensemble de N sources périodiques ou sporadiques $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ avec $\tau_i = \{T_i, C_i, D_i\}$ classées dans l'ordre non-décroissant des périodes (i.e. pour deux sources τ_i, τ_j, si $i < j$, alors $T_i \le T_j$) et $D_i = T_i$. Si τ est ordonnançable, on a :

C1: $$\sum_{i=1}^{N} \frac{C_i}{T_i} \le 1$$

C2: $\forall i,\ 1 < i \le N;\ \forall L,\ T_1 < L < T_i$:

$$L \ge C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{T_j} \right\rfloor C_j$$

Si τ satisfait les conditions C1 et C2 ci-dessus, alors NP-EDF peut ordonnancer n'importe quel ensemble concret généré à partir de τ. C'est à dire τ_i avec une date initiale r_i.

Le sens de C1 est clair. C'est la charge globale normalisée qui ne doit jamais dépasser 1. Une autre interprétation de C1 peut être que pour un intervalle de temps quelconque, la demande de traitement est toujours inférieure à la longueur de l'intervalle. C2 décrit une répartition extrême des flux d'arrivée : le client C_i occupe le serveur et tous les autres arrivent juste après une unité de temps (temps discret). Le serveur doit alors être capable de terminer le traitement de C_i, ainsi que le traitement des autres arrivées (représentées par le deuxième terme dans C2) sans dépasser une échéance.

Pour un ensemble τ dans le modèle MIQSS on peut utiliser ce théorème pour dimensionner la capacité de traitement du serveur c ($C_i = W_i/c$). Dans [Li03] un algorithme est développé pour trouver le c minimal.

En ce qui concerne la garantie déterministe (m,k)-firm dans le modèle MIQSS, si l'on considère (m,k)-WFQ comme un cas particulier de MIQSS et DWCS [West04] comme étant trop restreint, un seul autre résultat proposé par [Ramanathan99] existe pour le cas de κ-pattern fixe selon l'équation 3 que nous instancions ici dans le modèle MIQSS pour prendre en compte les sources multiples.

Pour une source $\tau_i = \{T_i, C_i, D_i, m_i, k_i\}$ le κ-pattern correspondant est une suite binaire de k_i bits $\prod_i = \{\pi_{i1}, \pi_{i2} \ldots \pi_{ik_i}\}$, qui satisfait : *(1)* le $n^{ème}$ client est critique si $\pi_{1(n\%ki)} = 1$ et optionnel si $\pi_{1(n\%ki)} = 0$; *(2)* $\sum_{j=1}^{k_i} \pi_{ij} = m_i$.

Le κ-pattern proposé dans [Ramanathan99] est donné par :

$$\pi_{ij} = \begin{cases} 1 & \text{Si } j = \left\lfloor \left\lceil \frac{j \times m_i}{k_i} \right\rceil \times \frac{k_i}{m_i} \right\rfloor \\ 0 & \text{Sinon} \end{cases} \qquad j = 1,2,\cdots k_i \tag{7}$$

Les demandes de traitement critiques sont ordonnancées selon RM (Rate Monotonic). La condition suffisante est donnée par le théorème suivant.

Théorème de [Ramanathan99] :

Considérons un ensemble de N sources périodiques ou sporadiques $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ avec $\tau_i = \{T_i, C_i, D_i, m_i, k_i\}$ classées dans l'ordre non-décroissant des périodes (i.e. pour deux sources τ_i, τ_j, si $i < j$, alors $T_i \le T_j$) et $D_i = T_i$. Définissons les termes ci-dessous :

$$R_{ij} = \left\{ \left\lfloor l \cdot \frac{k_i}{m_i} \right\rfloor T_j : \left\lfloor l \cdot \frac{k_i}{m_i} \right\rfloor T_j < T_i, l \in Z_+ \right\}$$

$$R_i = \bigcup_{j=1}^{i-1} R_{ij}$$

$$n_j(t) = \left\lceil \frac{m_j}{k_j} \left\lceil \frac{t}{T_j} \right\rceil \right\rceil$$

$$W_i(t) = C_i + \sum_{j=1}^{i-1} n_j(t) \cdot C_j$$

Si $min_{t \in R_i} W_i(t)/t \le 1$ pour tout $1 \le i \le N$, alors la politique RM respecte de façon déterministe toutes les contraintes (m_i,k_i)-firm.

Dans la pratique pour un ensemble de source $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ avec dates initiales quelconques, trouver la capacité du serveur c minimale requise pour la garantie déterministe selon ce théorème est NP-difficile [Quan00].

Dans [Quan00] des algorithmes heuristiques sont proposés. Afin de minimiser la charge instantanée dans le pire cas (qui permet de diminuer la demande en c), [Quan00] propose de répartir plus uniformément les m_i parmi les k_i en faisant la rotation des m_i selon l'équation suivante.

$$\pi_{ij} = \begin{cases} 1 \quad \text{si} \quad j - s_i = \left\lfloor \left\lceil \frac{((j-1)-s_i)\times m_i}{k_i} \right\rceil \times \frac{k_i}{m_i} \right\rfloor + 1 \\ 0 \qquad \text{sinon} \end{cases} \qquad (8)$$
$$j = 1, 2, \cdots k_i$$

où s_i est le nombre de périodes obtenues par le décalage circulaire vers la droite.

Un algorithme heuristique choisit une valeur de s_i provoquant ainsi moins d'interférence de demandes d'exécution par rapport à l'algorithme de [Ramanathan99]. Ce principe de rotation ne change pas de κ-pattern vis à vis d'une source mais change simplement la répartition dans l'axe du temps des demandes d'exécution critiques de N sources. En réalité, cette rotation veut introduire une sorte de κ-pattern dynamique. De ce point de vue DBP le fait plus facilement par l'affectation de priorité en-ligne.

4.2. Condition suffisante pour NP-DBP-EDF

Par rapport à NP-EDF dans [Jeffay91], NP-DBP-EDF introduit une variable supplémentaire qui est la valeur de DBP à l'instant t. Pour un client de la source τ_i sa priorité DBP est calculée selon l'équation 2 et on la note par $DBP_j(t)$. A un instant t, l'ensemble des clients peut être classé en trois classes suivantes :

1) Le client en cours de service dans le serveur
2) Les clients en attente avec $DBP_j(t) = 1$, i.e., ces clients doivent être exécutés par le serveur et terminer leur service avant leurs échéances respectives, sinon la garantie (m,k)-firm sera violée

2+*i*) Les clients en attente avec $DBP_j(t) = i$ (i>1), i.e., un tel client sera exécuté si le serveur est disponible et si l'exécution peut terminer avant son échéance, sinon il sera écarté par le serveur et le prochain client de la source aura sa priorité augmentée : $DBP_j(t+T_j) = DBP_j(t)-1$

Nous rappelons qu'en cas d'égalité de priorité DBP, EDF est utilisé.

La condition suffisante est donnée par le théorème suivant (cf. [Li03] pour la preuve).

Théorème de [Li03] :

Considérons un ensemble de N sources périodiques ou sporadiques $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ avec $\tau_i = \{T_i, C_i, D_i, m_i, k_i\}$ classées dans l'ordre non-décroissant des périodes (i.e. pour deux sources τ_i, τ_j, si $i < j$, alors $T_i \le T_j$) et $D_i = T_i$. Si τ satisfait les conditions C1 et C2 suivantes durant un intervalle de temps L quelconque, NP-DBP-EDF peut alors ordonnancer n'importe quel ensemble concret τ' généré par τ. C'est à dire qu'il n'y aura aucune violation de contrainte (m_i,k_i)-firm pour $i = 1, 2, \ldots, N$.

C1:
$$\sum_{i \in U} \left(\left\lfloor \frac{L}{k_i T_i} \right\rfloor m_i + Min\left(\left\lfloor \frac{L - k_i T_i \left\lfloor \frac{L}{k_i T_i} \right\rfloor}{T_i} \right\rfloor, m_i \right) \right) C_i +$$
$$\sum_{j \in \tau - U} \left(\left\lfloor \frac{L-1-(DBP_j(t)-2)T_j}{k_j T_j} \right\rfloor m_j + Min\left(\left\lfloor \frac{\left(L-1-(DBP_j(t)-2)T_j\right) - k_j T_j \left\lfloor \frac{L-1-(DBP_j(t)-2)T_j}{k_j T_j} \right\rfloor}{T_j} \right\rfloor, m_j \right) \right) C_j \le L$$

C2:
$$\forall i, \forall L, L > \min(T_i)$$
$$\left(\left(\left\lfloor \frac{L - C_i}{k_i T_i} \right\rfloor m_i + 1 \right) + Min\left(\left\lfloor \frac{L - C_i - \left\lfloor \frac{L - C_i}{k_i T_i} \right\rfloor k_i T_i}{T_i} \right\rfloor - 1, m_i - 1 \right)_+ \right) C_i +$$

$$\sum_{j\in\tau-\tau_i}\left(\left\lfloor\frac{L-1-(DBP_j(t)-2)T_j}{k_jT_j}\right\rfloor m_j+Min\left(\left\lfloor\frac{\left(L-1-(DBP_j(t)-2)T_j\right)-k_jT_j\left\lfloor\frac{L-1-(DBP_j(t)-2)T_j}{k_jT_j}\right\rfloor}{T_j}\right\rfloor,m_j\right)\right)C_j\le L$$

Où U est l'ensemble de clients de DBP = 1 qui peuvent arriver au même instant t et τ - U l'ensemble des autres clients. Dans le pire cas cet ensemble U peut inclure un client de chaque source et τ - U = $\varnothing$ (ensemble vide). Dans la pratique pour un ensemble concret τ', ce pire cas peut ne jamais être atteint.

En fait, cette expression de condition suffisante est celle de NP-EDF avec une variable qui est $DBP_j(t)$.

Nous avons démontré [Li03] par ailleurs que pour un système avec des valeurs de m_i et k_i quelconques (avec $m_i < k_i$, pour $i = 1, 2, \ldots N$ qui représente le numéro de source), cette condition suffisante peut être équivalente à la condition définie dans le cas du temps réel dur : (k,k)-firm.

Pour un ensemble concret de sources, un programme développé dans [Li03] peut être utilisé pour évaluer la différence en terme de demande de capacité de traitement du serveur entre (m,k)-firm et (k,k)-firm.

Figure 6 et Figure 7 montrent ce qu'on peut obtenir par ce programme pour le cas concret du Tableau 3. L'abscisse représente un intervalle de temps L et l'ordonnée la demande de serveur devant être exécutée avant la fin de l'intervalle de temps L (i.e. arrivée cumulative du travail). Dans chaque figure la courbe supérieure correspond à la demande de (k,k)-firm et celle inférieure correspond à la demande de (m,k)-firm. On a supposé que toutes les $DBP_i(t) = 1$ (le pire cas) pour (m,k)-firm.

	contrainte (m,k)	C_i	$T_i = D_i$
Source 1	(2,5)	8	12
Source 2	(4,5)	10	20
Source 3	(3,6)	2	5
Source 4	(1,5)	4	6

Tableau 3. Un cas concret du MIQSS

On peut constater que la demande de serveur de (m,k)-firm ne dépasse jamais celle de (k,k)-firm mais les deux courbes se superposent pour des petites valeurs de L.

Comment éviter cette situation indésirable constitue alors un objectif de nos travaux futurs car le dimensionnement du serveur du modèle MIQSS en dépend directement. Dans [Li03] une analyse des causes de la superposition est développée et nous concluons que la meilleure approche d'ordonnancement et les meilleurs κ-patterns doivent être donnés par le serveur (ordonnanceur). Ce qui peut être réalisé par l'établissement d'un protocole de négociation de la QdS entre les sources et l'ordonnanceur.

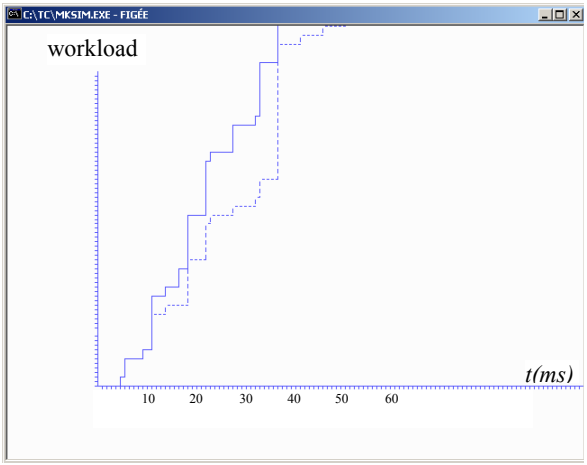


Figure 6. Différence en demande de serveur entre conditions 1 de [Li03] et de [Jeffay91]

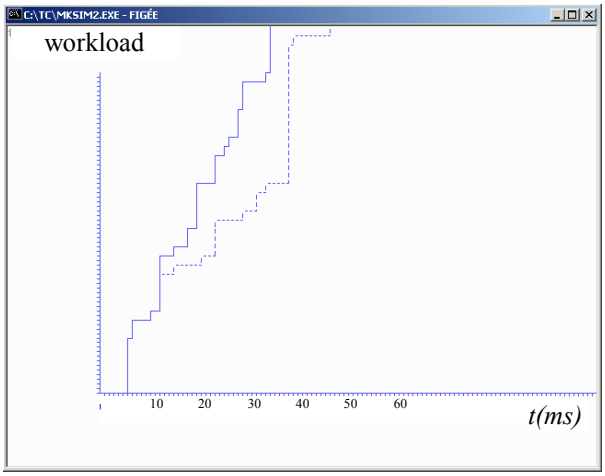


Figure 7. Différence en demande de serveur entre conditions 2 de [Li03] et de [Jeffay91]

5. Conclusion et perspectives

Offrir la QdS temps réel avec dégradation contrôlée selon le modèle (m,k)-firm consiste en une piste intéressante pour la conception des systèmes temps réel adaptatifs. En effet par rapport à la conception des lois de commande adaptatives en fonction de la variation de la QdS dans un système de contrôle-commande distribué, qui est basée sur une métrologie explicite en temps réel de la QdS du réseau [Michaut03], une approche utilisant par exemple DBP a l'avantage d'être simple car la « métrologie » de la QdS du système (certes se réduit au seul paramètre qui est équivalent à la charge) est réalisée implicitement par la k-séquence qui peut être considérée comme un historique de la QdS du réseau. Parmi les algorithmes d'ordonnancement sous contrainte (m,k)-firm, nous préférons les algorithmes dynamiques tels que DBP et DWCS aux algorithmes utilisant un κ-pattern fixe. Ceci pour essentiellement deux raisons : la capacité d'adaptation en-ligne à la variation d'état du système (variation en flux d'entrée, en capacité de traitement du serveur, …) et le potentiel de mieux utiliser le serveur dans le modèle MIQSS. Cette dernière est simple à comprendre. Considérons une source ayant déjà les *m* premiers clients servis. Le serveur, en cas de surcharge, peut ne pas servir les *k-m* clients suivants tout en satisfaisant la contrainte de (m,k)-firm. Tandis qu'avec un κ-pattern fixe, le serveur ayant déjà servi les *m* premiers clients (critiques et optionnels) risque de continuer à servir encore des clients s'il y des clients critiques dans les *k-m* clients suivants selon le κ-pattern.

Nos travaux futurs visent principalement deux directions : 1) Implémentation de la gestion dynamique de QdS selon le modèle (m,k)-firm dans les réseaux; 2) Recherche de conditions suffisantes d'ordonnançabilité avec d'autres algorithmes d'ordonnancement pour la garantie déterministe de (m,k)-firm ainsi que leur exploitation pour le dimensionnement du serveur dans le modèle MIQSS.

References

[ARTIST03] Project IST-2002-34820, Roadmap, "Adaptive Real-Time Systems for Quality of Service Management", http://www.systemes-critiques.org/ARTIST/Roadmaps/ , May 6th 2003.

[Bernat01] Bernat, G., A. Burns and A. Llamosi, "Weakly-hard real-time systems", *IEEE Transactions on Computer*s, 50(4), pp.308-321, April 2001.

[Bernat97] Bernat G. and A. Burns, "Combining (n, m)-hard deadlines and dual priority scheduling", *Proceedings of Real-Time Systems Symposium*, pages 46–57, Dec 1997.

[Chang00] Chang, C. S., *Performance Guarantees in Communication Networks*. New York: Springer-Verlag, 2000.

[Chow01] Chow, M.Y., Y. Tipsuwan, "Network-based control adaptation for network QoS variation", *IEEE Military Communications Conference (MILCOM2001)*, Vol. 1, pp257 -261, 2001.

[Chung90] Chung, J.Y., Liu, J.W. and Lin, K.J., "Scheduling periodic jobs that allows imprecise results", *IEEE Trans. on Computers*, 39(9):1156-1175, sep. 1990.

[El-Gendy03] El-Gendy, M.A., A. Bose, K.G. Shin, "Evolution of the Internet QoS and support for soft real-time applications", *proceedings of the IEEE*, Vol.91, No.7, pp1086-1104, July 2003.

[Furht99] Furht, B. (Editor), *Handbook of multimedia computing*, CRC Press LLC, 1999.

[Hamdaoui95] Hamdaoui M. and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines", *IEEE Transactions on Computer*s, 44(4), 1443–1451, Dec.1995.

[Hamdaoui97] Hamdaoui M. and P. Ramanathan, "Evaluating Dynamic Failure Probability for Streams with (m, k)-firm Deadline", *IEEE Transactions on Computer*s, 46(12), pp.1325–1337, Dec.1997.

[Jeffay91] Jeffay, K., Stanat, D.F. and Martel, C.U., "On Non Pre-emptive Scheduling of Periodic and Sporadic Task", *IEEE real-time systems symposium*, pp129-139, San Antonio (USA), Dec. 4-6, 1991.

[Jumel03] Jumel, F., N. Navet, F. Simonot-Lion, « Influence des performances d'une architecture informatique sur la fiabilité des systèmes échantillonnés ». *Edition Teknéa RTS'2003.* (Paris). 2003.

[Koubâa04a] Koubâa, A., Song, Y.Q., "Loss-Tolerant QoS using Firm Constraints in Guaranteed Rate Networks", *10th IEEE Real-Time and Embedded Technology and Applications (RTAS'2004)*, Toronto (Canada) 25-28 May 2004.

[Koubâa04b] Koubâa, A., *Gestion de la qualité de service temporelle selon la contrainte (m,k)-firm dans les réseaux à commutation de paquets*, Thèse de l'INPL, Nancy, 27 octobre 2004.

[LeBoudec02] Le Boudec, J.Y. and P. Thiran, *Network Calculus: A Theory of Deterministic Queueing Systems For The Internet*, Online Version of the Book of Springer Verlag – LNCS 2050, July 2002.

[Li03] Li, J., « Sufficient condition for guaranteeing (m,k)-firm real-time requirement under NP-DBP-EDF scheduling », *Rapport de DEA d'informatique de Lorraine*, LORIA, Juin 2003.

[Li04] Li, J., Y.Q. Song, F. Simonot-Lion, « Schedulability analysis for systems under (m,k)-firm constraints », *WFCS2004*, Vienna (Austria), Sept. 22-24, 2004.

[Martin04] Martin S., « *Maîtrise de la dimension temporelle de la Qualité de Service dans les réseaux* », Thèse Université Paris 12 (projet Hipercom INRIA), 6 Juillet 2004.

[Michaut03] Michaut F., « *Adaptation des applications distribuées à la Qualité de Service fournie par le réseau de communication* », Thèse UHP Nancy 1, 26 Nov. 2003.

[Nilsson98] Nilsson, J., et al, "Stochastic analysis and control of real-time systems with random time delays", *Automatica*, vol.34, pp.57-64, 1998.

[Parekh93] Parekh, A.K., R.G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case", *IEEE/ACM Transactions on Networking*, Volume 1, Issue 3, pp344-357, June 1993.

[Quan00] Quan G. and X. Hu, "Enhanced Fixed-priority Scheduling with (m, k)-firm Guarantee", *Proc. Of 21st IEEE Real-Time Systems Symposium*, pp.79-88, Orlando, Florida, (USA), November 27-30, 2000.

[Ramanathan99] Ramanathan P., "Overload management in Real-Time control applications using (m, k)-firm guarantee". *IEEE Transactions on Parallel and Distributed System*s, 10(6):549–559, June 1999.

[Wang02] Wang, S., Y. Wang, K. Lin, "Integrating Priority with Share in the Priority-Based Weighted Fair Queueing Scheduler for Real-Time Networks" *Journal of RTS,* p119-149, 2002.

[West04] West, R., Y. Zhang, K. Schwan and C. Poellabauer, "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers", *IEEE Transactions on Computers*, Volume 53, Number 6, pp. 744-759, June 2004.

[West99] West, R. and K. Schawn, "Dynamic window-constrained scheduling for multimedia applications", *6th International Conf. On Multimedia Computing and Systems, ICMCS'99*, IEEE, June 2000.

Qualité de Service dans les Réseaux Sans Fil

Guy Juanole
LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse cedex 4
juanole@laas.fr

Thierry Val
EA ICARE 3050
1 place Georges Brassens BP 60073
31703 Blagnac cedex
val@iut-blagnac.fr

Résumé

Les réseaux locaux sans fil sont de plus en plus utilisés actuellement. Le but de cet article est de présenter leurs aptitudes à offrir une Qualité De Service. Les trois principaux réseaux locaux sans fil sont analysés par rapport à ce critère, principalement au niveau de la couche MAC. Nous commençons par une présentation du LR-WPAN (Low Rate Wireless Personal Area Network) ZigBee. Dans une seconde partie, nous détaillons le WPAN Bluetooth. Enfin, nous terminons par le WLAN WiFi que l'on doit maintenant appeler ASFI en France ! La conclusion donne une idée des travaux de recherche potentiels à mener actuellement dans ce domaine.

1. Introduction

Les réseaux locaux sans fil sont, en 2005, employés massivement par le grand public pour des applications informatiques courantes : accès sans fil à INTERNET (ASFI !), domotique, bureautique, téléphonie, photographie numérique, applications audio-vidéo liées au home-cinéma… Dans le monde industriel également, de plus en plus de systèmes de communications sans fil sont employés pour des applications aussi diverses que la robotique mobile, la télé-surveillance, la télé-relève de compteurs, la sécurité des biens et des personnes, la télé-maintenance….

Le but de cette présentation est de faire le point sur les aptitudes de ces réseaux locaux sans fil à être utilisés pour des applications temps réels. En d'autres termes, les différentes couches protocolaires, et en particulier les couches basses, régissant ces réseaux sans fil, sont-elles basées sur des méthodes déterministes offrant une Qualité De Service ?

Afin de limiter le cadre de l'étude, nous focaliserons notre analyse sur les trois principaux réseaux locaux sans fil actuellement employés pour des communications sans fil de courte portée : ZigBee, Bluetooth et WiFi (ASFI).

2. Le réseau sans fil ZigBee

2.1. Présentation générale

ZigBee est une norme de Réseau personnel sans fil destiné à l'électronique embarquée à très faible consommation et bas débit (250 Kbps) associé à un faible taux d'utilisation du médium (typiquement moins de 1%) [2]. Le projet, initié en 1998, a conduit la ZigBee alliance à soumettre à l'IEEE une norme référencée 802.15.4. Les premiers circuits intégrés voient le jour actuellement. Les nœuds ZigBee, embarqués et alimentés par des piles pouvant durer plusieurs mois, équipent généralement des capteurs/actionneurs, et peuvent travailler dans plusieurs bandes radio : 868 MHz, 915 MHz et surtout 2.4 GHz. Deux types de nœuds sont prévus : les entités complètes, ou FFD pour *Full Function Device*, qui implémentent toutes les fonctionnalités protocolaires (représentée en blanc dans la figure 1) ; et les entités simplifiées, ou RFD pour *Reduced Function Device*, qui ne disposent que de fonctions réduites de base (représentée en bleu clair dans la figure 1). Une entité ayant des fonctionnalités potentielles de FFD peut très bien jouer le rôle de simple RFD (représentée en bleu foncé dans la figure 1).

2.2. Différentes topologies

Plusieurs topologies sont associées à une architecture de WPAN ZigBee. Dans la topologie en étoile (la plus courante) présentée figure 1, les entités sont reliées à un nœud central appelé le coordinateur du PAN, par lequel passent toutes les communications.

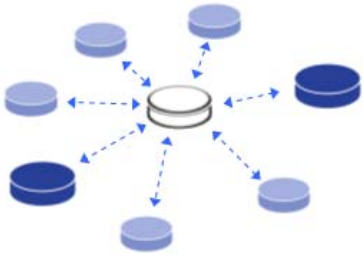


Figure 1. Topologie en étoile de ZigBee.

Dans la topologie point à point (la plus simple) illustrée figure 2, une entité peut communiquer directement avec toute autre entité à portée radio. Il faut évidement dans ce cas là, que chaque nœud soit à portée de tous les autres. On peut retrouver dans cette topologie des RFD et des FDD.

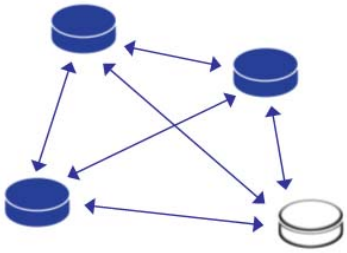


Figure 2. Topologie point à point de ZigBee.

Enfin, dans la topologie la plus aboutie (*Mesh network*), correspondant à une couche réseau ZigBee plus complexe, il est possible à plusieurs PAN de communiquer entre eux, ceci via leur coordinateur respectif, comme on le voit en figure 3.

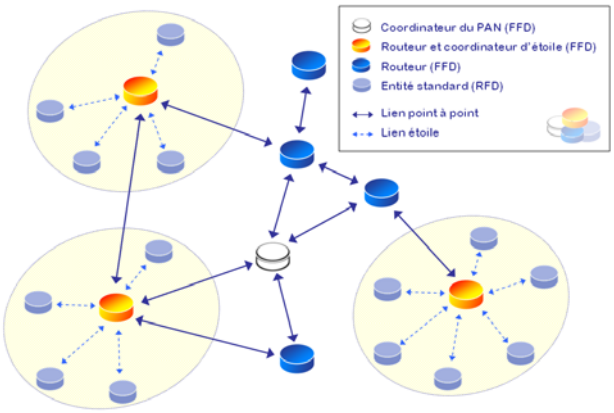


Figure 3. Topologie Mesh ZigBee.

2.3. La couche MAC de ZigBee

Dès le début des travaux, les concepteurs de ZigBee ont prévu une gestion possible de la QdS. Ceci se fait en grande partie grâce à une couche MAC (*Medium Access Control*) adaptée.

La couche MAC gère les accès au médium radio. 802.15.4 propose deux modes d'accès au médium : un mode non coordonnée (totalement CSMA/CA sans RTS/CTS) utilisable en particulier avec la topologie point à point, et un mode coordonné (*beacon mode*) utilisable avec la topologie en étoile, où le coordinateur envoie régulièrement des trames balises pour synchroniser les autres entités de son PAN. Le mode non coordonné est relativement classique et offre peu d'innovation par rapport à la version utilisée par WiFi (cf. §4). En revanche, le mode coordonné permet d'entrevoir des applications intéressantes mettant en oeuvre une QdS.

Dans le mode coordonné présenté figure 4, l'espace temporel entre deux trames balises est appelé supertrame. La structure de la supertrame est choisie par le coordinateur du réseau en fonction des demandes. La supertrame est divisée en 16 slots temporels de durée égale. La trame balise occupe toujours l'intégralité du premier slot de la supertrame : elle permet de diffuser une synchronisation sur tout le réseau, mais aussi l'identifiant du PAN et la structure de la suite de la supertrame. Le mode coordonné de 802.15.4 propose aussi deux "sous modes" à l'intérieur de la supertrame :

- un mode CSMA/CA (*Contention Access Period*) : les accès se font de façon classique, aléatoirement, tout en étant encadrés par une émission régulière de trames balises pour assurer la synchronisation entre les entités du PAN autour du coordinateur,
- un mode sans collision (*Contention Free Period*) : la structure de la supertrame est partiellement ou totalement maîtrisée par le coordinateur. L'accès au médium est alors organisé et réparti par le coordinateur et les collisions sont rendues impossibles (sauf en cas d'erreur de transmission, de forte mobilité des nœuds…).

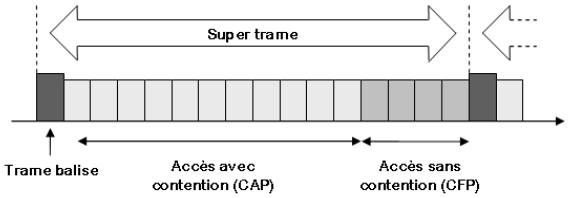


Figure 4. Supertrame ZigBee.

Ce dernier mode (CFP), rend possible une réservation de bande passante et peut offrir une certaine garantie sur le plan temporel. Pour cela, le coordinateur peut attribuer jusqu'à 7 créneaux garantis (*Guaranted Time Slot – GTS*), alors qu'un GTS peut occuper un ou plusieurs des 7 derniers slots de la supertrame. Les GTS sont placés à la fin de la supertrame dans la *CFP* : ainsi, le début de la supertrame *CAP*, reste en accès libre et aléatoire sans QdS par la méthode d'accès classique CSMA/CA pour permettre l'accès aux transports non garantis et aux nouvelles entités de se présenter sur le PAN.

3. Le réseau sans fil Bluetooth

3.1. Présentation générale

Bluetooth a été le premier réseau personnel sans fil. En effet, le but initial de ce WPAN est bien de remplacer les fils et les câbles autour des équipements électriques communicants. C'est initialement la société Ericsson qui a développé le projet, dans le but avoué d'offrir un moyen de communication local pour les téléphones portables, vers d'autres équipements informatiques pour les données, et vers une oreillette sans fil pour la voix. Ceci a entraîné, dès la conception protocolaire, l'existence de deux types de liens sans fil : les liens asynchrones (ACL) sans QdS pour le transport des données et les liens synchrones (SCO) avec QdS pour le transport de la voix numérisée à 64 Kbps dans les deux sens.

3.2. La couche radio de Bluetooth

La couche physique radio 2.4GHz de Bluetooth est basée sur l'étalement de spectre grâce à un saut de fréquence FHSS (*Frequency Hopping Spread Spectrum*) tous les 625µs (slots), présentée figure 5.

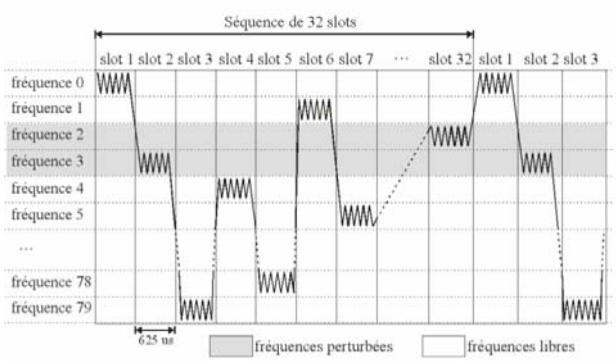


Figure 5. Principe du FHSS de Bluetooth.

Trois classes de puissance permettent des portées de 1, 10 et 100m. La modulation utilisée est la GFSK (*Gaussian Frequency Shift Keying*) et offre un débit nominal de 1 Mbps brut.

Associée au FHSS, la couche bande de base utilise une transmission TDD (*Time Division Duplex*), qui permet une communication bidirectionnelle alternative entre deux nœuds Bluetooth, comme nous pouvons le voir en figure 6.

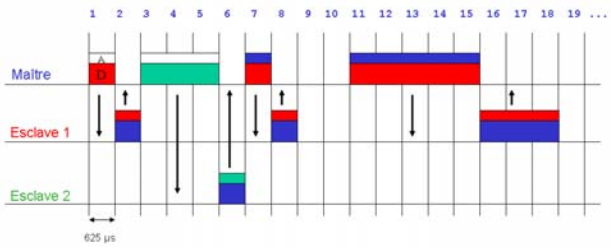


Figure 6. Principe du TDD de Bluetooth (Liens ACL).

3.3. Les topologies de Bluetooth

Ces caractéristiques physiques imposent une synchronisation entre les communicants. Ceci est assuré par un Maître dans le PAN qui synchronise tous ses Esclaves. Bluetooth se base alors sur trois topologies spécifiques, présentées figure 7.

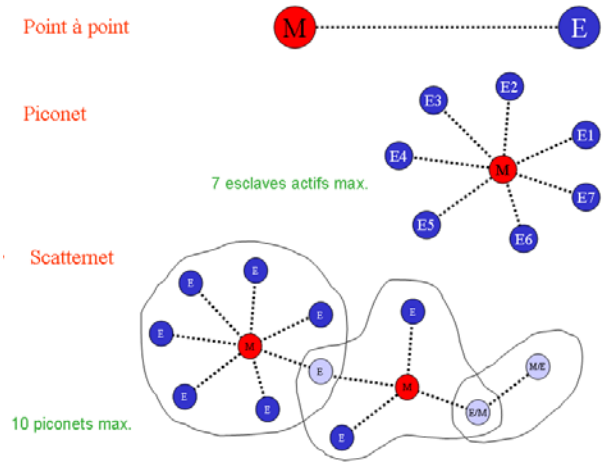


Figure 7. Les trois topologies de Bluetooth.

La plus simple est une topologie point à point entre un Maître et un Esclave. Le Maître choisit une séquence de sauts de fréquences et l'impose à son Esclave. Grâce au TDD, le Maître émet sur le premier slot vers son esclave qui lui répond obligatoirement sur le slot suivant. Dans une seconde topologie plus évoluée appelée *Piconet*, un Maître peut contrôler jusqu'à 7 Esclaves, qui suivant la nature des liens, seront interrogés plus ou moins cycliquement. Enfin, il est même prévu de pouvoir regrouper jusqu'à 10 *Piconets* entre eux afin de former un *Scarternet*, grâce à des nœuds relais entre *Piconets.*

3.4. Deux types d'applications : deux types de liens

Deux grands types d'applications potentielles ont conduit les initiateurs du projet Bluetooth à prévoir dès le début, deux catégories de liens, ceux principalement dédiés au transport de la voix numérisée, nommés SCO (*Serial Connection Oriented*), et ceux initialement prévus pour le transport de données des applications informatiques, nommés ACL (*Asynchronous Connection Less*).

3.5. Les liens asynchrones

Les utilisations typiques de cette dernière catégorie peuvent être :

- la liaison entre deux ordinateurs portables lors de transferts de fichiers,
- la synchronisation des fichiers entre un ordinateur de poche et un ordinateur de bureau,
- l'impression de documents sur une imprimante Bluetooth,
- la liaison sans fil entre un ordinateur mobile et un modem jouant le rôle de point d'accès Internet...

Dans tous ces cas, des volumes importants de données sont échangés. Les besoins en débit sont typiquement de 50 à 700 kbps, parfois dissymétriques, comme cela peut être le cas pour un accès 'http' entre un client mobile et un serveur fixe distant. Pour ces liaisons de données, la notion de la qualité de service n'est généralement pas nécessaire, pas plus que des latences de transmission faibles [4]. Ces liens sont donc asynchrones et sans connexion. Néanmoins, dès les premières spécifications de Bluetooth, il était prévu, au dessus des liens ACL de pouvoir agir sur des paramètres de QdS au niveau de la couche liaison L2CAP. Plusieurs paramètres sont prévus :

- *Service Type* qui identifie le niveau de service : *Best Effort* ou *Guaranteed*. Ce dernier est basé sur une interrogation cyclique des Esclaves par leur Maître,
- *Token Rate* qui permet de spécifier le débit de trafic moyen (en octets/seconde),
- *Token Bucket Size* qui spécifie la taille de la rafale (en octets),
- *Peak Bandwidth* qui spécifie le débit maximum autorisé pour la source (en octets/seconde),
- *Latancy* qui spécifie le délai maximum entre le moment ou le paquet est généré et son début de transmission au niveau radio (en microsecondes et en mode *Guaranted*),
- *Delay variation* qui spécifie la différence entre le minimum et le maximum du délai. Le récepteur se sert de ce paramètre pour fixer la taille de son buffer.

On retrouve là les notions et les mécanismes classiques utilisés pour offrir une Qualité de Service dans les réseaux filaires. Malheureusement, ces notions ne sont que très rarement implémentées dans les modules Bluetooth. De plus, ces fonctions de niveau L2CAP se reposent toujours sur des couches inférieures non déterministes. En effet, les liens ACL utilisent des codes détecteurs d'erreur de type CRC. Les retransmissions demandées automatiquement dans le cas d'erreurs détectées (mode ARQ pour *Automatic Repeat Query*) entraînent des retards non bornés. Le non déterminisme des liens ACL se retrouve également dans le cas de l'arrivée de nouveaux nœuds Bluetooth désirant s'insérer dans un Piconet ; ou dans le cas encore plus délicat du

Scaternet, où les changement de Maîtres induits par les relais de trames entre Piconet imposent aux Esclaves concernés des reconfigurations incessantes et lentes !

3.6. Les liens synchrones avec QdS native

En fait, seuls les liens synchrones offrent réellement une véritable QdS, associée à des applications à temps réel dur [1]. Pour les liaisons de type voix, un débit de 64 kbps symétrique assure une qualité d'écoute équivalente au standard ISDN (*Integrated Services Digital Network*) filaire. La qualité de service associée à cette catégorie requiert entre autre, une bande passante garantie, ainsi que l'absence de gigue dans les retards de propagation et d'accès au médium. Ces fortes contraintes temporelles sont satisfaites par le caractère synchrone de ces liaisons associées à un mode orienté connexion. Cette première catégorie de liens est nommée SCO pour *Serial Connection Oriented*. La période T_{sco} d'accès au canal est respectivement de 2, 4 et 6 fois 625 µs pour les liens HV1, HV2 et HV3 (*High quality Voice 1-2-3 slots*), comme nous pouvons le voir dans les figures 8, 9 et 10.

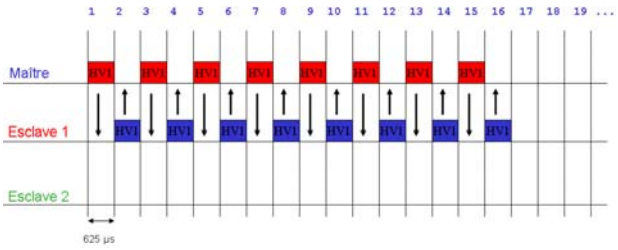


Figure 8. Lien SCO paquet HV1.

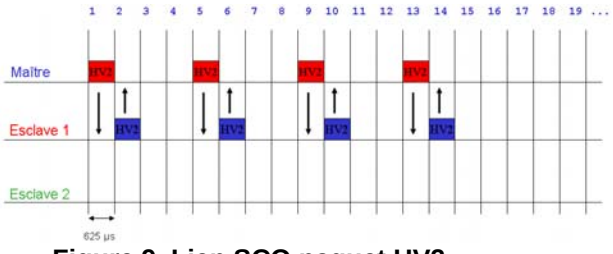


Figure 9. Lien SCO paquet HV2.

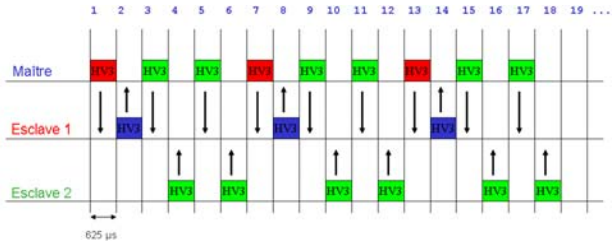


Figure 10. Lien SCO paquets HV3.

Ainsi, ces trois liens HVi offrent le même débit utile de 64 kbps, bien qu'ils n'offrent pas la même longueur de champ de données utiles, à cause de la différence de redondance induite par des codes correcteurs différents (1/3 FEC ou 2/3 FEC pour *Forward Error Correction*). Les signaux analogiques vocaux sont numérisés au moyen d'un code PCM-CVSD (*Pulse Code Modulation-Continuous Variable Slope Delta Modulation*) avec un échantillonnage sur 1 bit à 64 Khz.

3.7. Les dernières évolutions de Bluetooth

Depuis les dernières versions de Bluetooth (V1.2 et surtout V2.0), d'importantes améliorations sont apparues, en particulier le saut de fréquence est maintenant adaptatif (AFH pour *Adaptatif Frequency Hopping*), ce qui permet d'éviter à un Piconet de travailler sur les mêmes fréquences occupées que d'autres réseaux sans fil comme WiFi par exemple. Ceci améliore grandement la qualité de la couche physique radio de Bluetooth et ne peut qu'aller dans le sens d'une meilleur QdS offerte par les couches supérieures. Les débits sont également supérieurs et sont de l'ordre de 1 Mbps utile. Enfin, de nouveaux profiles sont apparus, tels que les profiles A/V qui sont dédiés au transport de l'audio et même la vidéo sur des canaux ACL. Là encore, la QdS n'est pas assurée parfaitement sur ces liens asynchrones, et d'importants temps de transmission sont à prévoir (plusieurs dizaines de millisecondes [3]), empêchant ainsi toute utilisation pour des applications interactives de qualité (téléphonie) ou surtout de contrôle-commande en milieu industriel.

4. Le réseau sans fil WiFi

Le réseau WiFi (*Wireless Fidelity*) est basé sur la norme IEEE 802.11 relative à la couche physique et à la couche liaison. Cette norme est le standard de base qui offre deux débits à 1 et 2 Mbits/s dans la bande 2.4 GHz et qui définit principalement la fonctionnalité MAC. La famille des standards 802.11 n'a pas cessé de s'agrandir avec des propositions de nouveaux standards [6] pour la couche physique et également la fonction MAC. En ce qui concerne cette dernière, la norme IEEE 802.11e a été définie pour intégrer des mécanismes de Qualité de Service.

Dans ce chapitre, nous allons nous focaliser sur la fonctionnalité MAC (norme IFFE 802.11 et norme IEEE 802.11e). Au préalable, nous indiquons encore que deux topologies peuvent être considérées : la topologie avec infrastructure (le réseau sans fil consiste en un ensemble d'équipements sans fil en lien radio

avec un point d'accès connecté à un réseau filaire ; cette configuration est appelée BSS (*Basic Service Set*) ; le point d'accès (AP : *Access Point*) est l'élément central qui est l'intermédiaire pour les communications entre les équipements sans fil) ; la topologie ad hoc, appelée encore IBSS (*Independant Basic Service Set*), est un ensemble d'équipements sans fil qui communiquent directement entre eux (pas d'élément central).

4.1. La fonction MAC (IEEE 802.11)

4.1.1. Généralités

A. Deux méthodes d'accès sont définies : la « *Distribution Coordination Fonction* » (DCF) qui consiste en un accès distribué aléatoire (influence de IEEE 802.3 mais avec une spécificité des liaisons sans fil (on ne peut pas écouter quand on émet, ce qui donne la technique CSMA-CA) ; la « *Point Coordination Function* » (PCF) qui consiste en un accès centralisé, géré par le point d'accès (AP), et qui est basé sur une interrogation à tour de rôle (*Polling*) des éléments sans fil par le AP.

Notons que la deuxième méthode (PCF) ne concerne que la topologie avec infrastructure. En ce qui concerne la première méthode (DCF), elle comprend deux modalités : la modalité de base CSMA-CA (*Carrier Sense Multiple Access Collision Avoidance*) et la modalité CSMA-CA avec échanges de trames courtes (RTS (*Request to Send*)–CTS (*Clear to Send*).

B. Le fonctionnement du MAC, quelle que soit la méthode, est basé sur le concept d'intervalles temporels fondamentaux i.e des intervalles temporels qui définissent des priorités d'accès au medium.

Trois intervalles temporels sont définis : le *Short Inter Frame Space* (SIFS), le *Point Coordination Inter Frame Space* (PIFS) et le *Distributed Inter Frame Space* (DIFS). On a les relations suivantes : SIFS $<$ PIFS$<$DIFS. Les valeurs dépendent de la couche physique et plus précisément des débits supportés [6]. Le PIFS n'existe pas dans la configuration ad hoc.

C. Les échanges dans la topologie avec infrastructure sont organisés suivant une succession de supertrames délimitée par des trames balises (*Beacon Frame*).

La figure 11 représente la structure d'une supertrame avec la trame balise, la partie CFP (*Contention Free Period*) qui concerne la méthode d'accès PCF et la partie CP (*Contention Period*) qui concerne la méthode d'accès DCF. Notons que par rapport à la supertrame ZygBee les parties CFP et CP sont inversées.

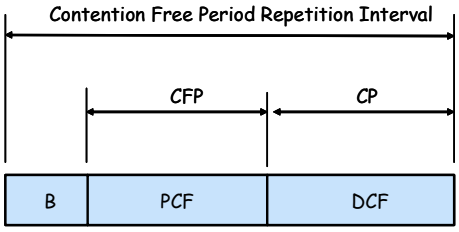


Figure 11. Structure de la supertrame

D. Le déploiement des réseaux sans fil pose le problème dit du « nœud (station) caché » qui se produit lorsque deux équipements sans fil ne peuvent s'entendre directement (soit du fait d'une distance trop grande, soit à cause de la présence d'un obstacle entre eux). La résolution de ce problème nécessite des mécanismes de communication particuliers (qui sont indiqués par la suite).

4.1.2. La méthode DCF

4.1.2.1. La modalité de base CSMA-CA

A. Le principe de base du comportement des éléments (équipements) sans fil voulant émettre est le suivant :

- tout d'abord il y a une écoute du medium
- ensuite deux cas sont à considérer

- Premier cas (medium libre pendant un temps DIFS) : on applique la procédure dite de « *backoff* » avant d'émettre.
- Deuxième cas (medium libre pour une durée inférieure à DIFS (et ensuite occupé) ou médium occupé) : on attend que le medium redevienne libre pendant un temps DIFS et on applique la procédure dite de « *backoff* ».

B. La procédure de « *backoff* » consiste à poursuivre l'observation du medium libre pendant une durée aléatoire (appelée « temps de *backoff* »). Si le medium reste libre pendant toute cette durée, alors une transmission est effectuée dès la fin de cette durée. Si, par contre, le medium devient occupé à un instant de cette durée, l'équipement concerné arrête le compteur mesurant l'écoulement de cette durée, attend que le medium redevienne libre (pendant DIFS) pour reprendre l'écoulement de cette durée jusqu'à sa fin où une transmission est alors effectuée. Notons que le processus

d'arrêt de l'écoulement du « temps de *backoff* » peut intervenir plusieurs fois. La durée du « temps de *backoff* » est un multiple du « *slot time* » qui est déterminé de manière aléatoire par chaque équipement, dans un intervalle [0, CW] où CW représente la fenêtre de contention « *Contention Window* ».

C. La modalité d'accès induit (par définition) des possibilités de collisions et comme (contrairement aux réseaux filaires) on ne peut pas mettre en oeuvre la technique d'écoute tout en émettant, le moyen de détecter qu'il n'y a pas de collision est d'obtenir un accusé de réception (ACK) de l'équipement récepteur. Lorsque l'équipement récepteur reçoit une trame, il attend, pour envoyer ACK, un temps SIFS (SIFS<DIFS) ce qui le rend le plus prioritaire par rapport à l'envoi d'autres trames par d'autres stations et donc garantit le transfert sans collision du ACK. Un exemple de transmission réussie à la première tentative est donné sur la figure 12. Si l'équipement émetteur d'une trame ne reçoit pas le ACK, il retransmet en mettant en oeuvre la procédure de « *backoff* ». La fenêtre de contention CW croît au fur et à mesure des retransmissions : on a la relation CW= CW_{min} (2b-1), CW_{min} est la valeur minimale utilisée lors de la transmission, b est une variable initialisée à 1 (lors de la première transmission) et incrémentée à chaque retransmission.

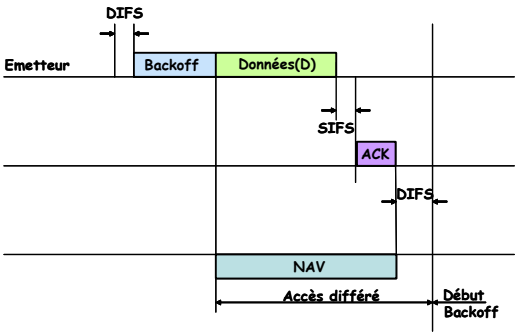


Figure 12. Exemple de transmission

D. Après chaque transmission réussie (c'est à dire on a la réception du ACK), l'entité MAC réalise une procédure « *backoff* », même s'il n'y a pas de nouvelle trame à envoyer. Cette procédure est encore appelée « *post backoff* » car le « *backoff* » est fait après une transmission et non avant une transmission.

Les exceptions à la règle (Faire la procédure « *backoff* ») avant une transmission sont :

1- A l'initialisation du réseau

2- Si une demande du niveau supérieur (LLC) pour envoyer une trame arrive à l'entité MAC quand :
- la file d'attente est vide
- le dernier « *post backoff* » a été effectué
- le medium est libre pendant DIFS et donc on peut transmettre à la fin de DIFS.

Notons que la procédure « *post backoff* » garantit qu'il y a toujours un temps aléatoire entre deux échanges consécutifs de trame.

E. Les collisions induisent des pertes dans l'utilisation de la capacité du medium et, ce d'autant plus que la longueur des données transférées dans une trame est grande. On peut fragmenter ces données afin de réduire les conséquences des collisions.

La modalité classique (DIFS + procédure « *backoff* ») est utilisée pour le premier fragment. Tous les autres fragments sont transmis sans interruption possible (SIFS et pas de procédure « *backoff* »).

F. Le champ durée des trames transférant des données ou des fragments de données permet aux autres stations de connaître pour quelle durée (durée de trame + SIFS + durée ACK) le medium est utilisé et donc d'ajuster leur NAV (*Network Allocation Vector).* Le NAV est un mécanisme important qui indique à chaque station la durée qui doit s'écouler jusqu'à ce que la transmission actuelle se termine et donc à quel instant elle pourra tester le medium.

4.1.2.2. La modalité CSMA-CA avec échange de trames courtes RTS et CTS

Les techniques des trames courtes RTS et CTS apportent des solutions tout d'abord à la problématique des stations cachées. Par exemple, si deux stations cachées A et B veulent envoyer une trame en un même temps à une troisième station C située à l'intersection de leur zone de couverture, comme elles ne s'entendent pas elles vont s'autoriser à le faire et il y aura donc collision ce qui empêchera C de recevoir les transmissions de A et B. L'utilisation de la trame RTS par les stations A et B diminue, du fait que la trame est courte, la probabilité de collision. Si la trame RTS est reçue par la station C, celle-ci envoie CTS, informant ainsi toutes les stations qui sont dans sa zone de couverture ce qui évite donc des tentatives de transmissions « presque » simultanées. Notons le rôle important du NAV

déduit de la lecture des champs durée des trames RTS et CTS. Notons que la technique des trames RTS et CTS est également intéressante pour résoudre le transfert de longues trames de données (nécessité de fragmentation, le processus RTS-CTS est associé au transfert du premier fragment).

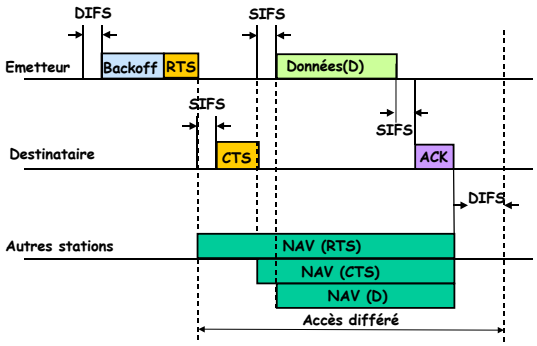


Figure 13 . Exemple d'utilisation de RTS/CTS

4.1.3. La méthode PCF

Les éléments principaux du fonctionnement de cette modalité d'accès sont résumés sur la figure 14. Notons l'utilisation de la temporisation PIFS pour que le PC puisse débuter la nouvelle supertrame (envoi de la trame balise) et la temporisation SIFS utilisée entre tous les échanges durant CFP. Les trames balises des supertrames doivent être normalement générées à des intervalles réguliers (TBTT : *Target Beacon Transmission Time*).

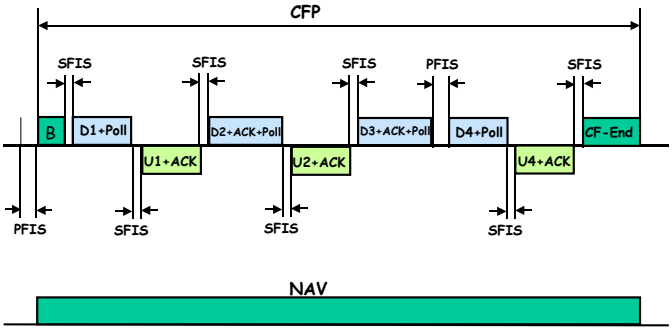


Figure 14. CFP et échanges

4.1.4. Problémes relatifs à la Qualité de Service

On ne peut garantir que deux trames Beacon consécutives sont distantes de TBTT (toute station peut émettre une trame dans la partie DCF même si la durée de cette trame peut aller au-delà de la valeur TBTT). Le temps de transmission des stations interrogées dans la partie CFP n'est pas spécifié. Dans la partie CP, il est impossible de distinguer les flux utilisateurs.

4.2. La proposition IEEE 802.11e

Cette proposition est une extension de la norme IEEE 802.11 (toujours la notion de supertrame avec la trame balise et les parties CFP et CP). Elle introduit deux nouvelles méthodes : la méthode EDCA (*Enhanced Distributed Channel Access*) utilisée dans la partie CP qui améliore la méthode DCF ; la méthode HCCA (*Hybrid Controled Channel Access*), utilisée, à la fois, dans la partie CFP et dans la partie CP, et mise en œuvre par *l'Hybrid Coordination Function* (HCF) du composant HC (*Hybrid Coordinator*) qui, de manière identique au PC, dans le cas de la norme IEEE 802.11, réside dans l'élément AP (*Access Point*).

4.2.1. Méthode EDCA

Le support de la Qualité de Service est réalisé au moyen de l'introduction de classes de trafic (*Traffic Categories* : TC) qui ont chacune une file d'attente associée et des paramètres spécifiques (AIFS (TC) c'est à dire l'*Arbitration Interframe Space* qui joue le rôle du DIFS de la norme IEEE 802.11 pour chaque file ; la fenêtre de contention CW (TC)).

Quatre TC sont différenciées : TC-VO (*voice*), TC VI (*video*), TC-BE (*Best Effort*), TC-BK (*Background*). Chaque TC est gérée par une entité en parallèle (cf. figure 15). Les valeurs des AIFS (TC) et des CW (TC) permettent de fixer [6] des priorités entre les classes de trafic (un exemple est donné sur la figure 16).

Il est important de noter que l'on peut avoir des collisions virtuelles (collisions entre les différentes files d'un équipement). Celle qui a la plus grande priorité est la gagnante pour faire la compétition sur le medium avec des files des autres équipements (cette compétition induit les collisions réelles).

En ce qui concerne les collisions virtuelles, les conséquences (pour les files non gagnantes)

sont identiques à une collision réelle (doublement de la fenêtre de contention).

Un autre attribut important de la proposition IEEE 802.11e est le paramètre TXOP (*Transmission Opportunity*) qui est un intervalle de temps (défini dans un champ de la trame balise) durant lequel une entité backoff peut transmettre plusieurs données venant du niveau supérieur.

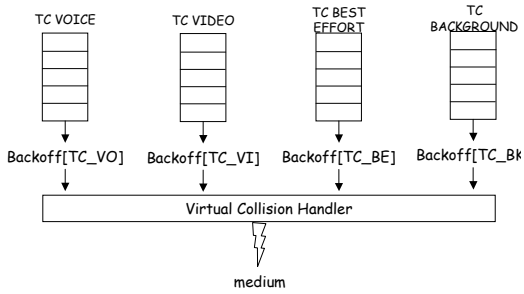


Figure 15. Les quatre files d'attente

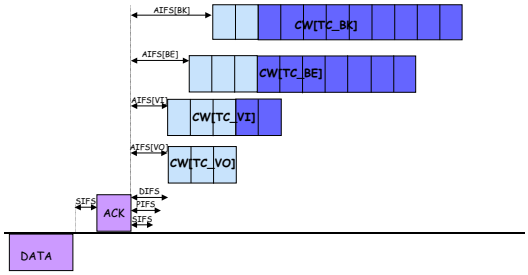


Figure 16.Système de priorités

4.2.2. Méthode HCCA

Le HCC attribue par *polling* dans les parties CFP et CP des opportunités de transmission (TXOP) à toute station (y compris la station AP). Notons que dans la partie CP, nous avons maintenant la cohabitation entre les techniques CSMA-CA et *polling.*

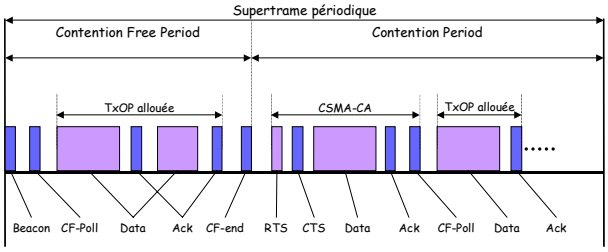


Figure 17. Echanges avec la méthode HCCA

La figure 17 représente un exemple d'échanges.

5. Conclusion

Alors que les premiers réseaux sans fil ne prenaient pas en compte les aspects déterministes de leurs couches protocolaires, on remarque que les nouvelles versions de WiFi et les nouveaux réseaux personnels sans fil tels que Bluetooth et surtout ZigBee offrent la possibilité aux applications temps réels d'utiliser ces médiums immatériels tout en conservant un certain respect de contraintes temporelles. Les méthodes d'accès proposées sont souvent basées sur des réservations de ressources simples, et il est dommage que les demandes de réservation ne soient généralement pas déterministes. De plus, la QdS est généralement présente uniquement si la mobilité des équipements sans fil n'entraîne pas de changement de cellules radio. L'attribution de l'accès au médium radio est souvent déterminée à priori, et peu d'adaptation et de variation de bande passante sont possibles. Les contraintes temporelles sont souvent respectées, à condition que ces dernières soient dans des valeurs supérieures à la milliseconde. Néanmoins, il est alors possible d'utiliser les réseaux sans fil pour le transport de données à contraintes temporelles, par exemple pour des flux audio et vidéo. Egalement, la voix sur IP via un WLAN est une application tout à fait envisageable aujourd'hui ! Les applications industrielles telles que la robotique sans fil ou la télé-maintenance sont également possibles sur des réseaux sans fil WiFi, ZigBee ou Bluetooth. Pourtant, il reste encore beaucoup de travaux de recherche à accomplir pour proposer :

- des couches physiques fiables, rapides et constantes,
- des méthodes d'accès déterministes même dans les phases d'initialisation, de pannes ou de forte mobilité,
- des couches LLC capables de corriger sans retard les erreurs de transmission,
- des couches 3 aptes à déterminer rapidement le meilleur chemin dans un réseau sans fil sans cesse en mouvement, et entraînant, à cause de sa topologie complexe et nécessairement redondante, une multitude de chemins possibles,
- des applications ne se "voilant plus la face", et nécessairement adaptables à des médiums sans fil aux capacités toujours plus réduites face aux médiums filaires classiques, et surtout aux caractéristiques pouvant changer très rapidement.

- Des méthodes formelles de spécification et de validation des protocoles utilisés dans les réseaux locaux sans fil [5].

Références

[1] A. van den Bossche, “Etude et réalisation d'un système de communication à QdS basé sur Bluetooth pour l'instrumentation sans fil appliqué à l'avionique”, *Rapport de stage d'ingénieur ESEO / DEA LIP6*, Laboratoire ICARE, 2004.

[2] A. van den Bossche, "Le LR-WPAN ZigBee", *Séminaire L2I*, Laboratoire LIRMM-CNRS, 15 mars 2005.

[3] A. van den Bossche, T. Val, E. Campo, " Métrologie pour l’analyse comparative des performances temporelles des liens Bluetooth", *IEEE SETIT 2005, International Conference : Sciences of Electronic, Technologies of Information and Telecommunications*, 27-31 Mars 2005, Sousse, TUNISIE.

[4] T. Val, G. Juanole, "Développement d'Applications de Métrologie pour le WPAN Bluetooth", *Colloque Francophone sur l'Ingénierie des Protocoles CFIP'2002*, Montréal, 27-30 mai, 2002.

[5] T. Khoutaif, F. Peyrard, G. Juanole, T. Val, "Vers l'utilisation des réseaux de Pétri pour l'évaluation des performances d'un système de communication sans fil dédié à la robotique", *Congrès National de Recherche dans les IUT, CNRIUT'2003*, Tarbes, 14 et 15 mai 2003.

[6] H. Labiod, H. Afifi « De Bluetouth à Wi-Fi », Hermès 2004.

Institut National Polytechnique de Lorraine

Impressions et Reliures :
INPL – Atelier de reprographie
2, avenue de la forêt de la Haye
B.P. 3. – F-54501 Vandoeuvre Cedex
Tel : 03.83.59.59.26 ou 03.83.59.59.27

Editeur : Nicolas NAVET, LORIA-INRIA

ISBN : 2 905267-47-X